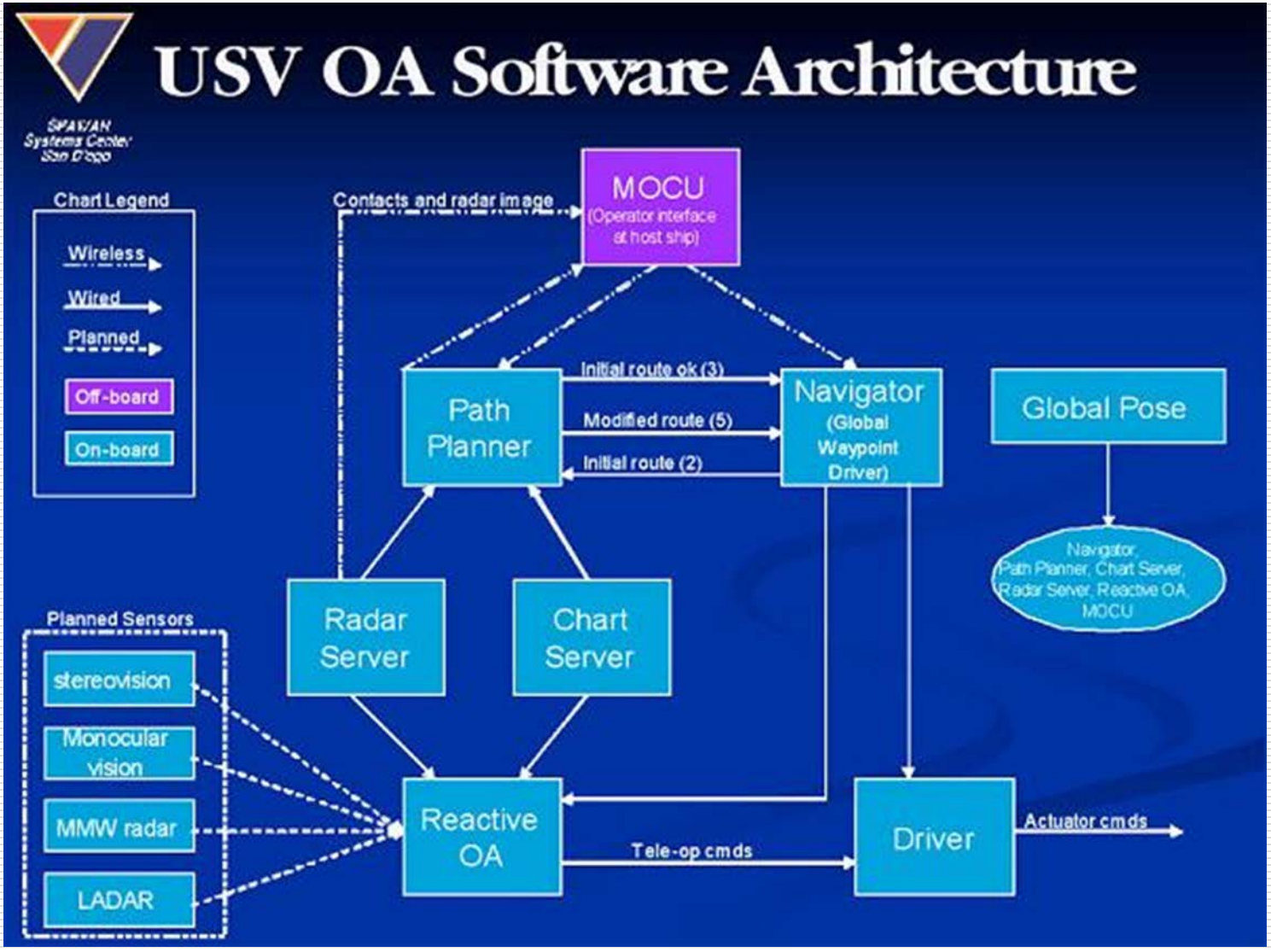
# Dimensions of Software Design

Andy Podgurski
EECS Department
Case Western Reserve University

# Software Design

- Activities after requirements specification and before coding:
  - Deciding on *structure* and *dynamics* of solution to *design problem* posed by SRS, including:
    - *System components*
    - *Relationships* and *interactions* between components
  - Documenting this in a *design document*
- Distinction from implementation is not always clear

# Example: Unmanned Surface Vehicle Software Architecture

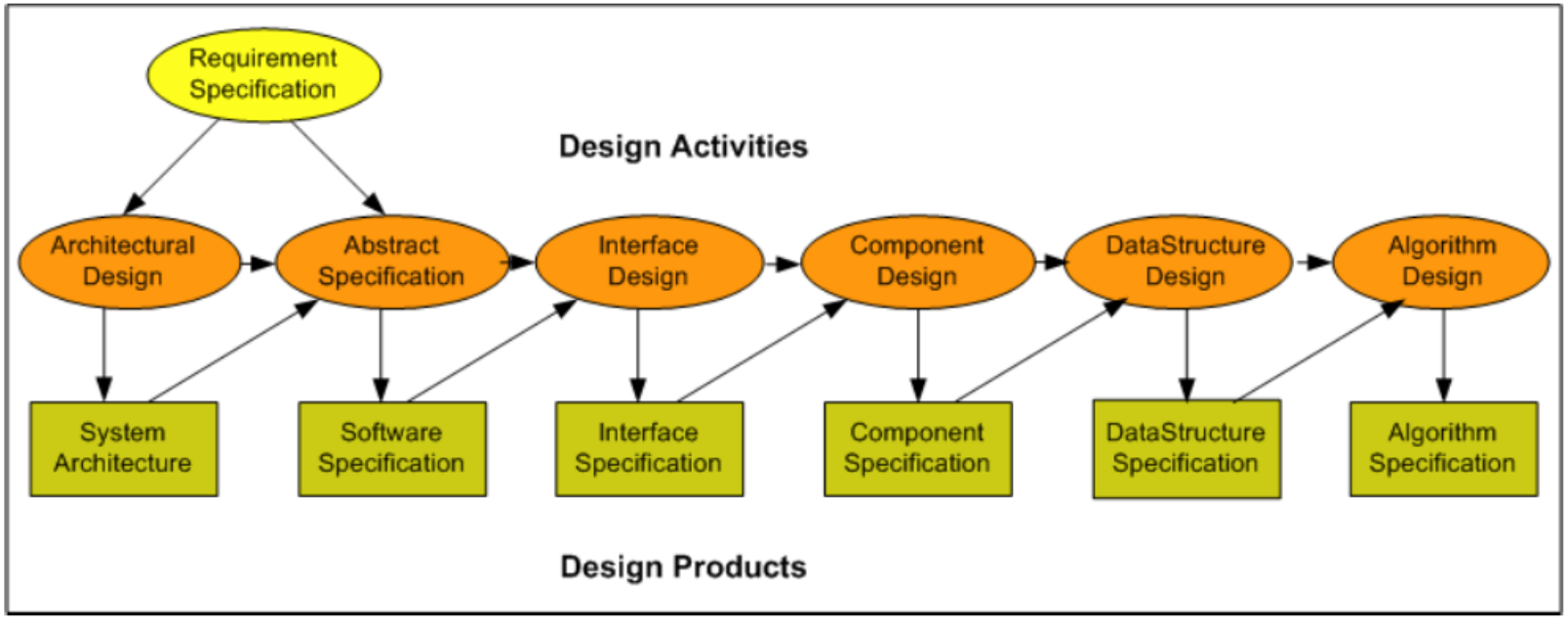



www.public.navy.mil/spawar/Pacific/Robotics/Pages/USV_OA.aspx

# Dimensions of Design

- Problem solving
- Modeling
- Specification
- Organizing the solution
- Communication
- Economics and reuse
- Maintenance and evolution

# Example of Design Process




www.cs.colorado.edu/~kena/classes/5828/s10/presentations/softwaredesign.pdf

# Design as Problem Solving

- Design problem is often very *complex*
- May involve making hundreds or thousands of *design decisions*
- Design is a *creative* process
- However a *strategy* is needed for coping with complexity

# Divide-and-Conquer

- Most important problem solving strategy
- Design problem is *decomposed*
- Designer shows how it can be solved by solving a set of simpler *design subproblems*
- Subproblems can be worked on *independently*

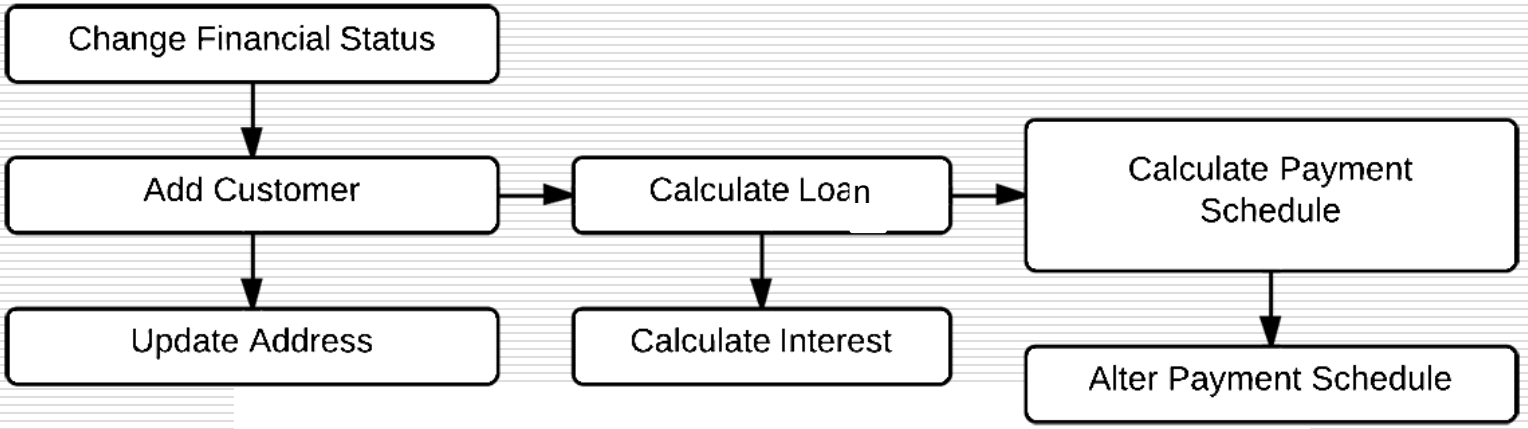
# Decomposing a Design Problem

- Designer acts as if there are *components* available that solve the subproblems
- He/she specifies how the original problem can be solved *using the components*
- This can be done by writing *code or pseudocode* that invokes the components

# Decomposition cont.

- Design and implementation of the components can be *deferred* until later
- If subproblems are complex, divide-and-conquer is applied *recursively* to them
- Design decomposition permits a project manager to *allocate* implementation work to multiple programmers
- Other names: *functional decomposition, top-down design, stepwise refinement*
  - See sunnyday.mit.edu/16.355/wirth-refinement.html

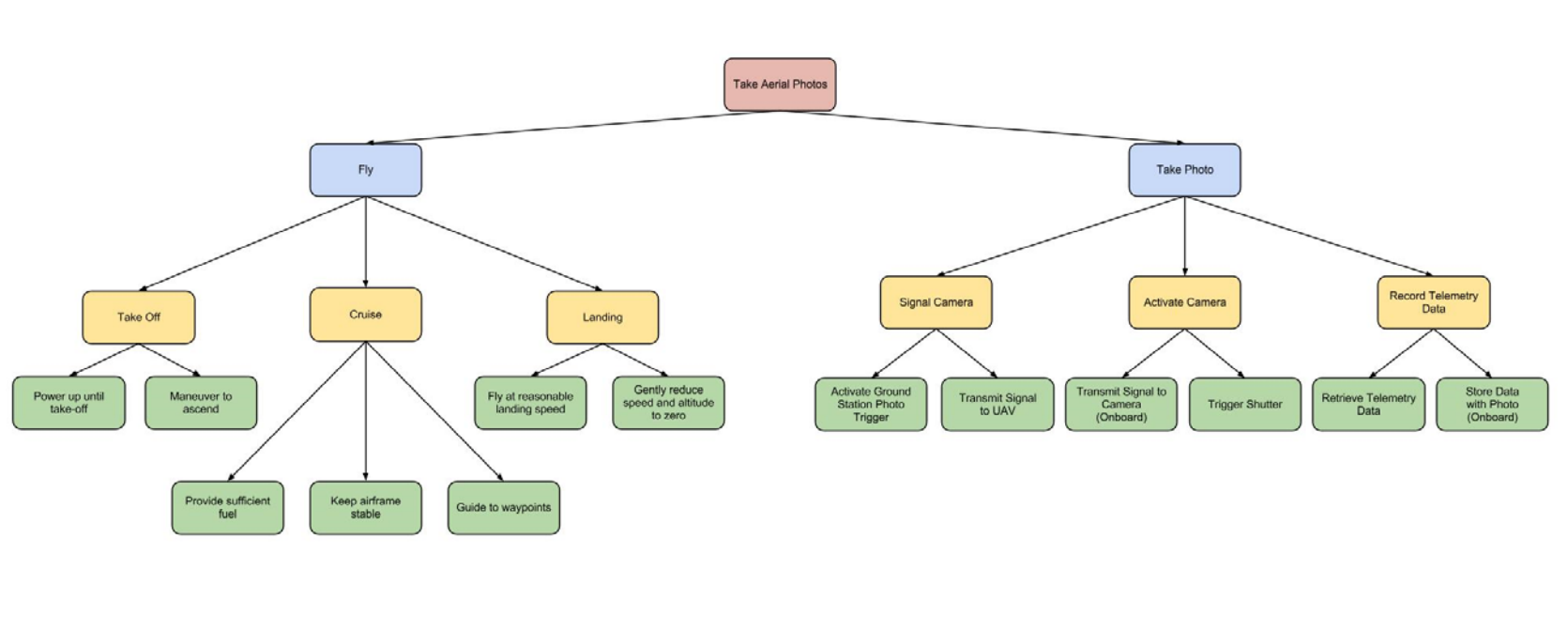
# Example: Decomposition of Customer Loan Processing




1.Adding a new customer.
2.Updating a customer address.
3.Calculating a loan to a customer.
4.Calculating the interest on a loan.
5.Calculating a payment schedule for a customer loan.
6.Altering a payment schedule.

From sourcemaking.com/antipatterns/functional-decomposition

# Example: UAV Aerial Imaging System Design




edge.rit.edu/edge/P14231/public/Nodes/Systems%20Design

# Procedural Design

- Popular before GUIs and object-oriented programming
- Design is conceived of as *set of procedures* that are composed
  - Sequentially
  - Conditionally
  - Iteratively
  - Hierarchically (via calls)
- Many design problems are amenable to procedural solutions

# Example: Payroll Application Pseudocode

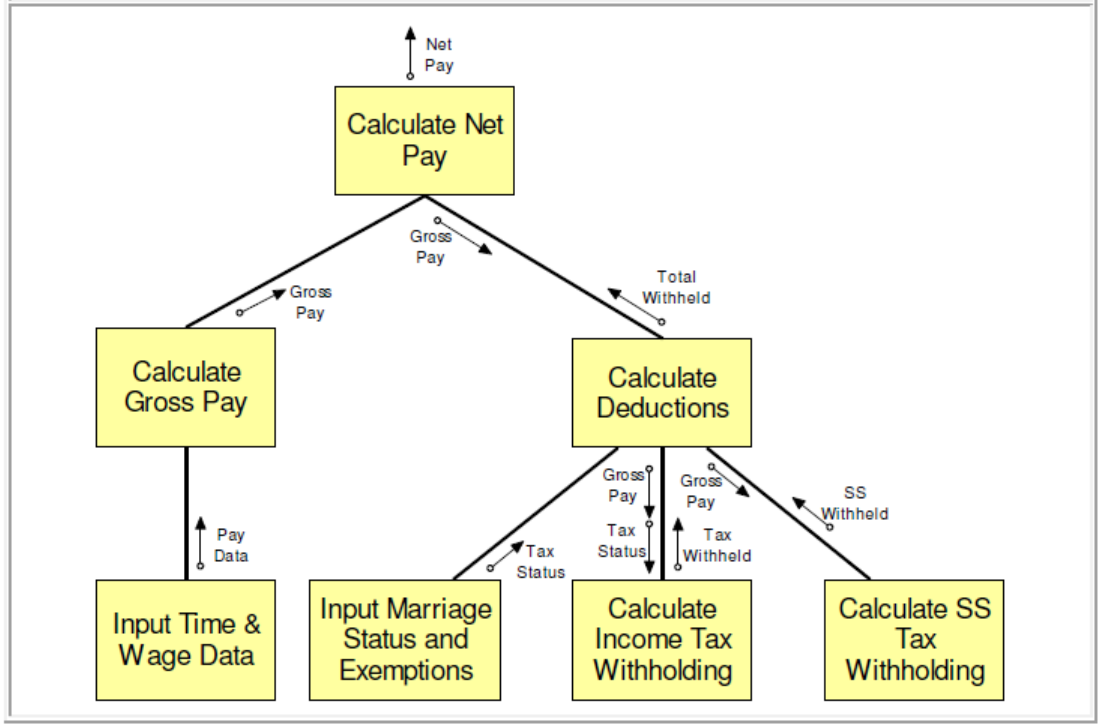
- Employees processed in order of their SSNs:

```
for (each employee e) {
    Get e's data.
    Calculate e's gross pay.
    Calculate e's tax withholding.
    Calculate e's net pay.
    Generate a payment to e.
    Generate a tax report for e.
}
```

# Example: Structure Chart

*Example*

This structure chart shows the hierarchical relationship between the methods within a computer program. Here the line symbol represents a method call and the data arrows represent arguments and return values. For instance, it shows that the program has a method to *Calculate Deductions* that receives the *Gross Pay* as an argument and returns the *Total Withheld*.




www.sal.ksu.edu/faculty/tmertz/Java/210proceduralprogramdesign/02proceduraldesignmethodology.pdf

# “Borderline” Procedural Designs

- In some applications, the main steps are *not fixed in advance*, e.g.:
  - Text editing, word processing
  - Drawing
  - Games
- User has *choice* of operations with *few constraints* between them
- Procedural designs for such applications have *main loop* to determine next operation
- Structure of code does *not* clearly reflect essential *information flows*
- This is a step toward *event-driven design*

# Example: Procedural Design for Text Editor

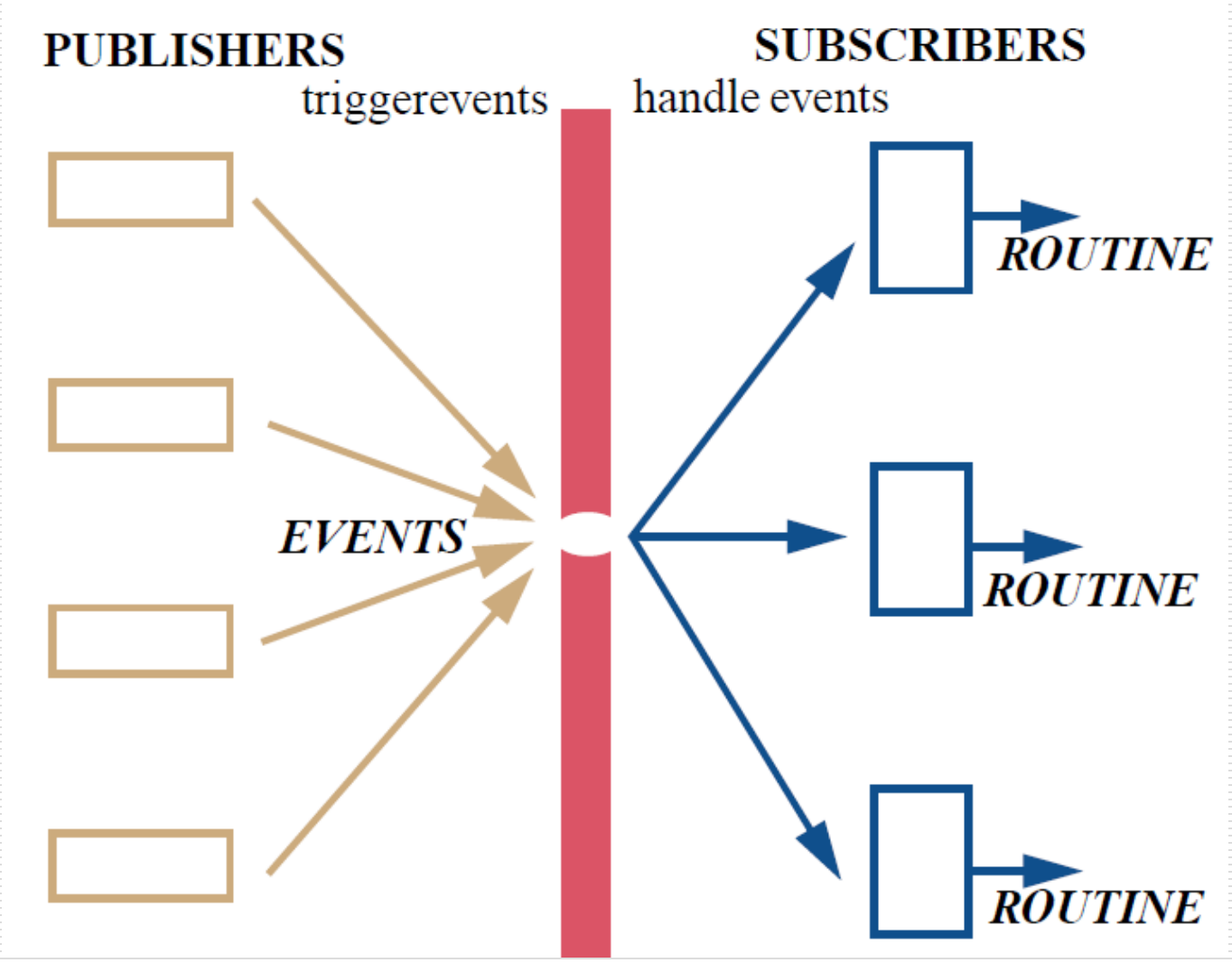
```
while (!quit) {
      op = getNextOperation();
      switch (op) {
            case INSERT:
              doInsert();
              break;
            case SELECT:
              doSelect();
              break;
            case CUT:
              doCut();
              break;
            case COPY:
              doCopy();
              break;
            case PASTE:
              doPaste();
              break;

            ...

            case EXIT:
              quit = true;
              break;
      }
}
```

# Event-Driven Design

- Appropriate when there is an *external source of events* to which the application must respond, e.g.,
  - Real-time systems
  - GUIs
- Programmer provides *event handlers*
- Events can occur and be handled *concurrently*

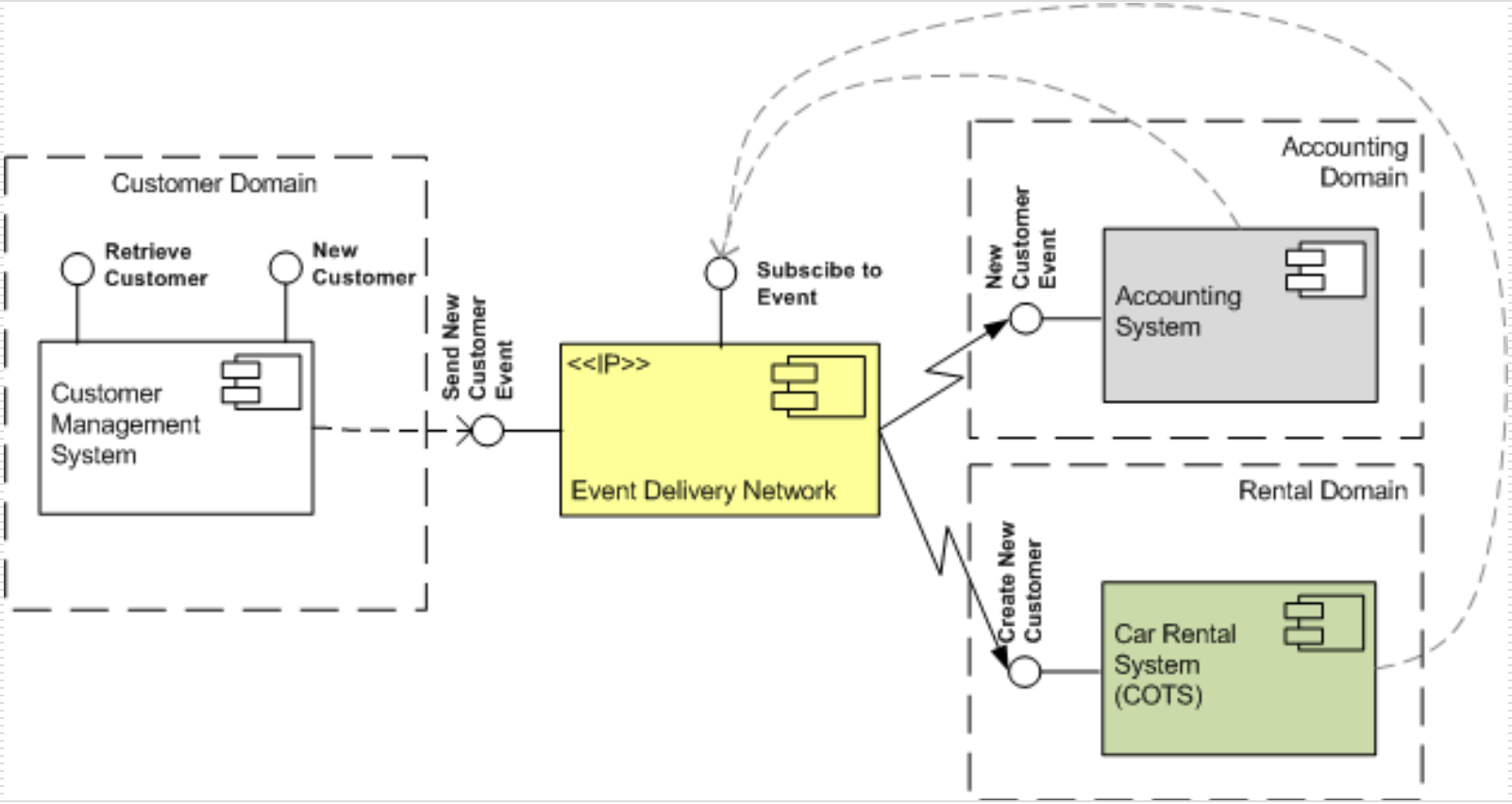
# Event-Driven Design cont.

- Event-driven apps force designers away from purely procedural thinking
- They must think in terms of events and event handlers or *listeners*
- Components interested in certain events *subscribe* to receive them




[Bertrand Meyer, *A Touch of Class*, touch.ethz.ch

# Example: Event-Driven Design




www.oracle.com/technetwork/articles/soa/schmutz-soa-eda-405955.html

# Example: Java *WindowListener* Interface

`void` **`windowActivated`**`(WindowEvent e)`
Invoked when the Window is set to be the active Window.

`void` **`windowClosed`**`(WindowEvent e)`
Invoked when a window has been closed as the result of calling dispose on the window.

`void` **`windowClosing`**`(WindowEvent e)`
Invoked when the user attempts to close the window from the window's system menu.

`void` **`windowDeactivated`**`(WindowEvent e)`
Invoked when a Window is no longer the active Window.

`void` **`windowDeiconified`**`(WindowEvent e)`
Invoked when a window is changed from a minimized to a normal state.

`void` **`windowIconified`**`(WindowEvent e)`
Invoked when a window is changed from a normal to a minimized state.

`void` **`windowOpened`**`(WindowEvent e)`
Invoked the first time a window is made visible.

**Figure : Java WindowListener interface**

# Event-Driven Designs cont.

- Responding to an event may require *complex processing*
- *Divide-and-conquer* can be used to solve this design subproblem
- It's common for *events* to be *decomposed* into lower-level events

# Example: Hierarchy of GUI Events

- Completion of data entry form →
  - TEXT_ENTERED, LIST_ITEM_SELECTED, BUTTON_PRESSED, ... →
    - MOUSE_DOWN, MOUSE_UP, KEY_DOWN, KEY_UP ...

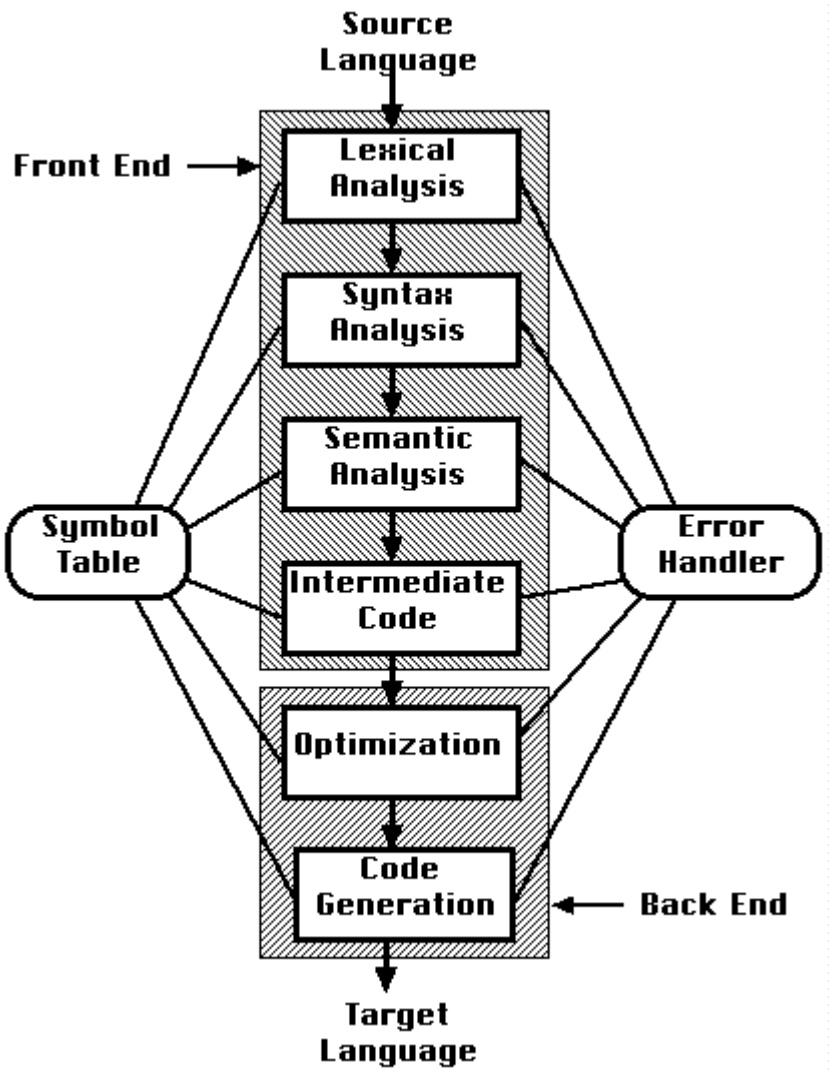
# Considering Alternative Designs

- There's usually more than one plausible way to solve a design problem
- Sometimes a "standard" architecture (high-level design) is available
- If not, designer must *explore alternatives*

# Example: Standard Architecture for Compiler

- Lexical analysis
- Syntax analysis
  - Top-down or bottom up
  - Table-driven or recursive descent
  - Numerous alternative algorithms
- Semantic analysis
- Intermediate code generation
- Optimization
- Object code generation

# Standard Compiler Architecture




www.cs.uaf.edu/~cs631/notes/phases.gif

# Evaluating Alternative Designs

- *Identify* reasonable alternatives
- *Sketch* each one
- *Discuss* with other developers
- Identify *advantages* and *disadvantages*
- If a clearly superior choice does not emerge, *prototype* leading candidates

# Recognition-Primed Decision Model [Klein]

1. Used by *expert* decision makers in different fields
2. A *tentative plan* comes to mind by automatic function of *associative memory*
3. Plan is *mentally simulated* to see if it will work
4. If plan seems *appropriate*, it is *implemented*
5. If it has *shortcomings*, it is *modified*
6. If it cannot be easily modified, *next most plausible option* is considered ...

# Design Quality

- To effectively compare alternative internal designs it's necessary to choose *quality factors* to consider
- *Quality metrics* are used to measure achievement of quality factors
  - These should be identified and documented during requirements analysis and specification, if possible (Why?)

# General Quality Factors

- Completeness
- Efficiency
- Maintainability
- Portability
- Reliability
- Security
- Simplicity
- Understandability

# Application-Specific Quality Factors

- Depend on application requirements, e.g.,
  - The most important quality factors for a sorting algorithm are correctness and computation time and space
  - The most important factor for a disease diagnosis algorithm is accuracy of diagnosis

# Quality Metrics

- Some metrics can be *computed* by analyzing *formal design descriptions*
  - E.g., computing degree of coupling between modules using module dependence graph
- Others can be calculated from *data* obtained when *exercising prototype*
  - E.g., speed of typesetting algorithm in pages/second
- Other metrics involve *subjective judgment*
  - E.g., visual quality of typeset pages
  - Can be estimated from *user ratings*
- *Maintainability* cannot be fully assessed prior to implementation

# Prioritizing Subproblems

- Useful heuristic:
  - *First attack subproblems that entail the greatest perceived risks, e.g., those least understood*

# Organizing the Design

- A complex design may involve hundreds or thousands of components
- These must be *organized* in an understandable way
- The organization of components influences the *amount of work* required to modify them

# Modularization

- Creation of program *modules*
- Module consists of *co-located components*
  - e.g., procedures or classes
- Module has an *interface* and an *implementation*
- Source code for one module is stored in one file or directory
- Modules can be compiled separately
- Well conceived modules often can be *reused*

# Coupling and Cohesion

- Modules that exhibit *low coupling* and *high cohesion* are easier to maintain
  - Coupling: tendency for changes to one module to entail changes to another
  - Cohesion: degree of conceptual relatedness of module's elements

# Encapsulation and Information Hiding

- A well-designed module *encapsulates* one or more design decisions
- It is said to *hide information* about them
- Data items are *bundled* with the operations that access them
- "Client" code uses the module only via its *interface*
- This avoids unnecessary coupling
  - E.g., direct access to underlying data structure

# Object-Oriented Design (OOD)

- Assumption: organizing a design around the *objects* in a system makes it easier to maintain
- An object has *state* and *behavior*
  - State of object consists of the values of a set of *attributes* it maintains
  - Behavior of object is embodied in a set of *methods* (functions) it provides for the use of other objects

# Example: MP3 Player Object

- State:
  - Playlist
  - Current file
  - Current operation
  - Current file position
  - Volume
  - Play time
  - Display appearance
- Behavior:
  - *addFiles*
  - *displayPlaylist*
  - *chooseFile*
  - *removeFile*
  - *play*
  - *pause*
  - *stop*
  - *previous*
  - *next*
  - *increaseVolume*
  - *decreaseVolume*
  - *save*
  - *exit*

# Class

- A declaration that permits creation of objects with the same *attributes* and *operations* (*methods*)
- An object is called an *instance* of class
- Is the basic unit of *modularization* and *reuse* in OOD

# Example: Java Class

```
public class SavingsAccount {

        private double balance;

        public SavingsAccount(double initialBalance, double initialInterest) {
                balance = initialBalance;
                interest = initialInterest;
        }

        public void deposit(double amount) {
                balance = balance + amount;
        }

        public void withdraw(double amount) {
                balance = balance - amount;
        }

        public void addInterest() {
                balance = balance + balance * interest;
        }

        public double getBalance() {
                return balance;
        }

}
```

From gist.github.com/jimmykurian/1732868

# Object-Oriented Analysis (OOA)

- *Initial phase* of object-oriented design
- Involves identifying classes of *real-world objects* application deals with
- *Services* they will provide to clients are determined and specified
- Detailed design of services may lead to creation of *new classes*

# Specifying the Design

- Required behavior of components must be *specified* so they can be developed or acquired later
- Initially, *informal* natural language definitions suffice
- Later, each component's *interface and behavior* must be specified *precisely*

# Example: Class, Responsibility, and Collaboration (CRC) Cards

Class: **Editor**

| Responsibilities: | Collaborators: |
| --- | --- |
| •Initialize editor.<br>•Manage editing actions.<br>•Manage files and buffers.<br>•Manage editor modes (customizations for editing particular file types).<br>•Manage plugins.<br>•Manage properties (configuration data).<br>•Manage user settings.<br>•Manage editor views.<br>•Exit on request. | •Buffer<br>•EditAction<br>•Mode<br>•View |

**Figure : CRC Card for text editor class**

# Example: Interface Specification for *jEdit* method *openFile*

```
public static Buffer openFile(View view,
                              java.lang.String parent,
                              java.lang.String path,
                              boolean newFile,
                              java.util.Hashtable props
```

Opens a file. This may return null if the buffer could not be opened for some reason.

**Parameters:**

`view` - The view to open the file in
`parent` - The parent directory of the file
`path` - The path name of the file
`newFile` - True if the file should not be loaded from disk be prompted if it should be reloaded
`props` - Buffer-local properties to set in the buffer

**Since:**

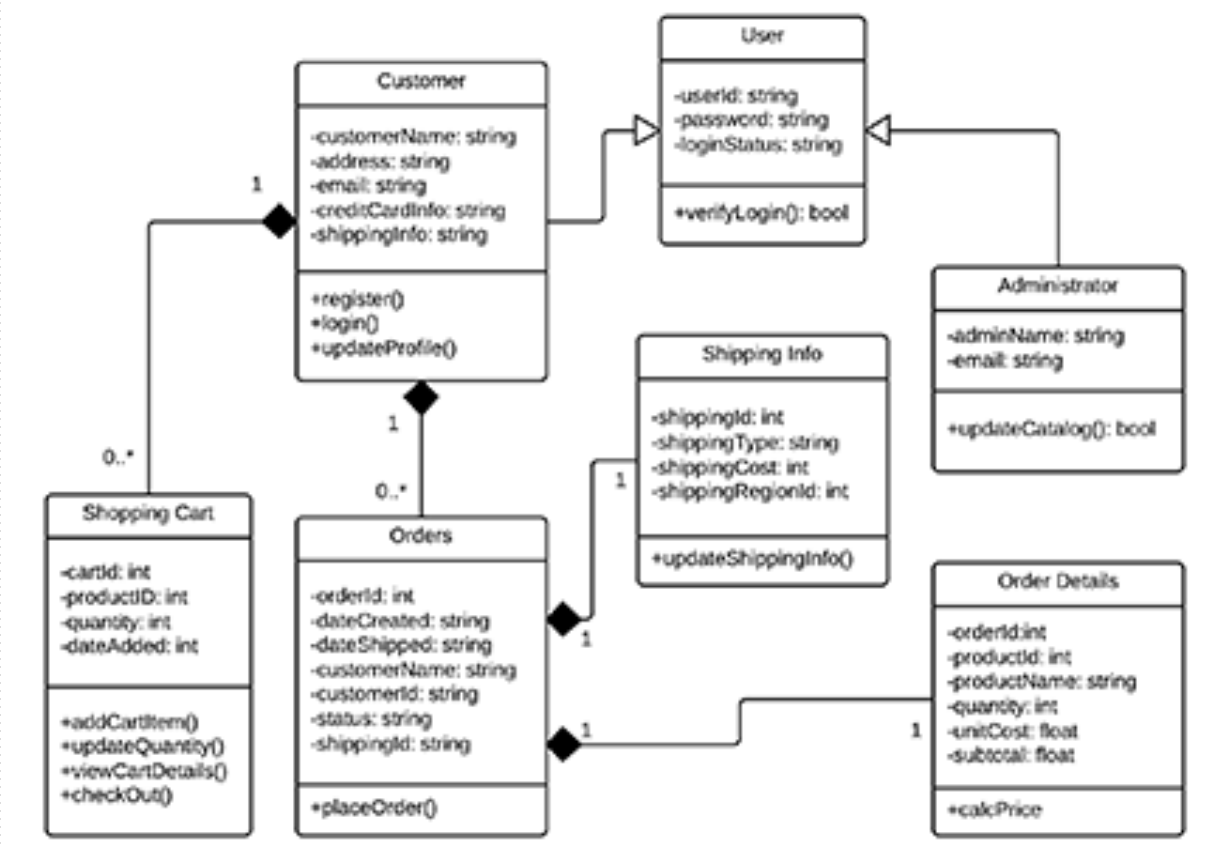
jEdit 3.2pre10

**Figure : Interface specification for method *jEdit.openFile***

# Modeling the Design

- A design model depicts a design's components and their relationships
- *Graphical models* are used to aid understanding and communication
- They *supplement* prose documentation, pseudocode, and interface specs
- A variety of model types are commonly used in software design, e.g,
  - Object and class diagrams
  - Data flow diagrams
  - Finite-state machine diagrams

# Example: Class Diagram




www.lucidchart.com

# Example: UML Sequence Diagram

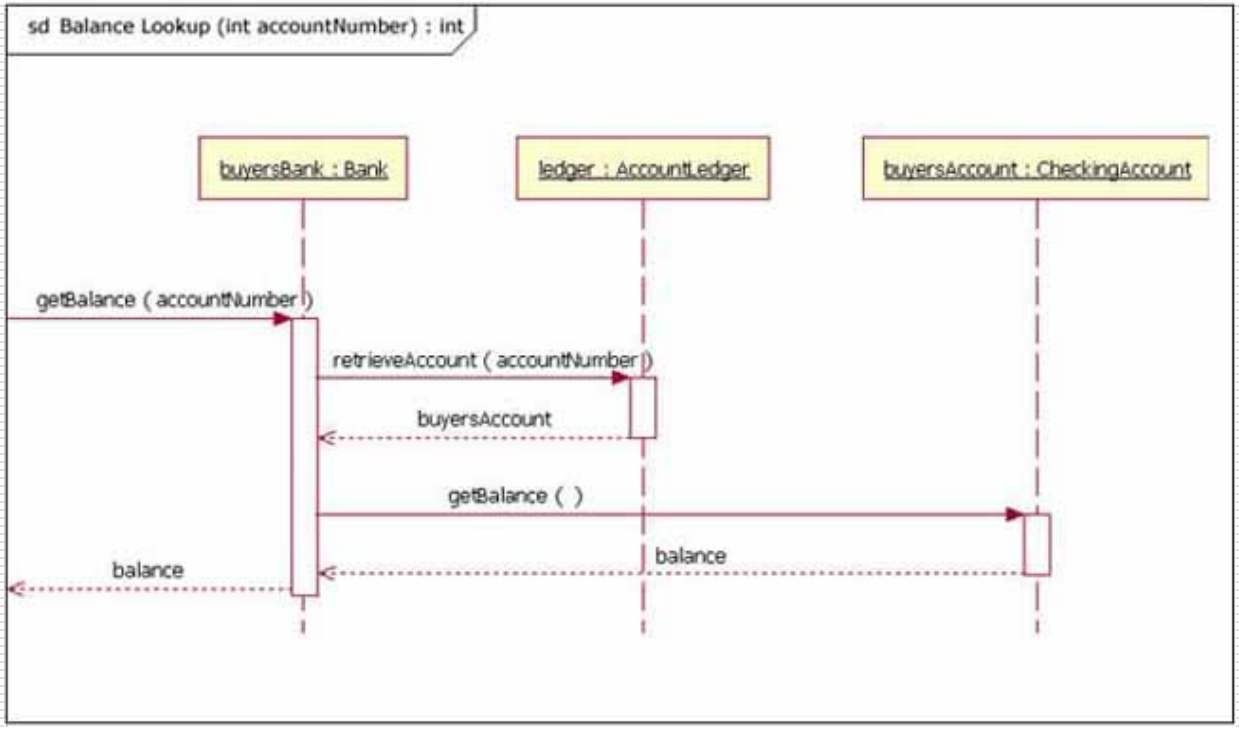


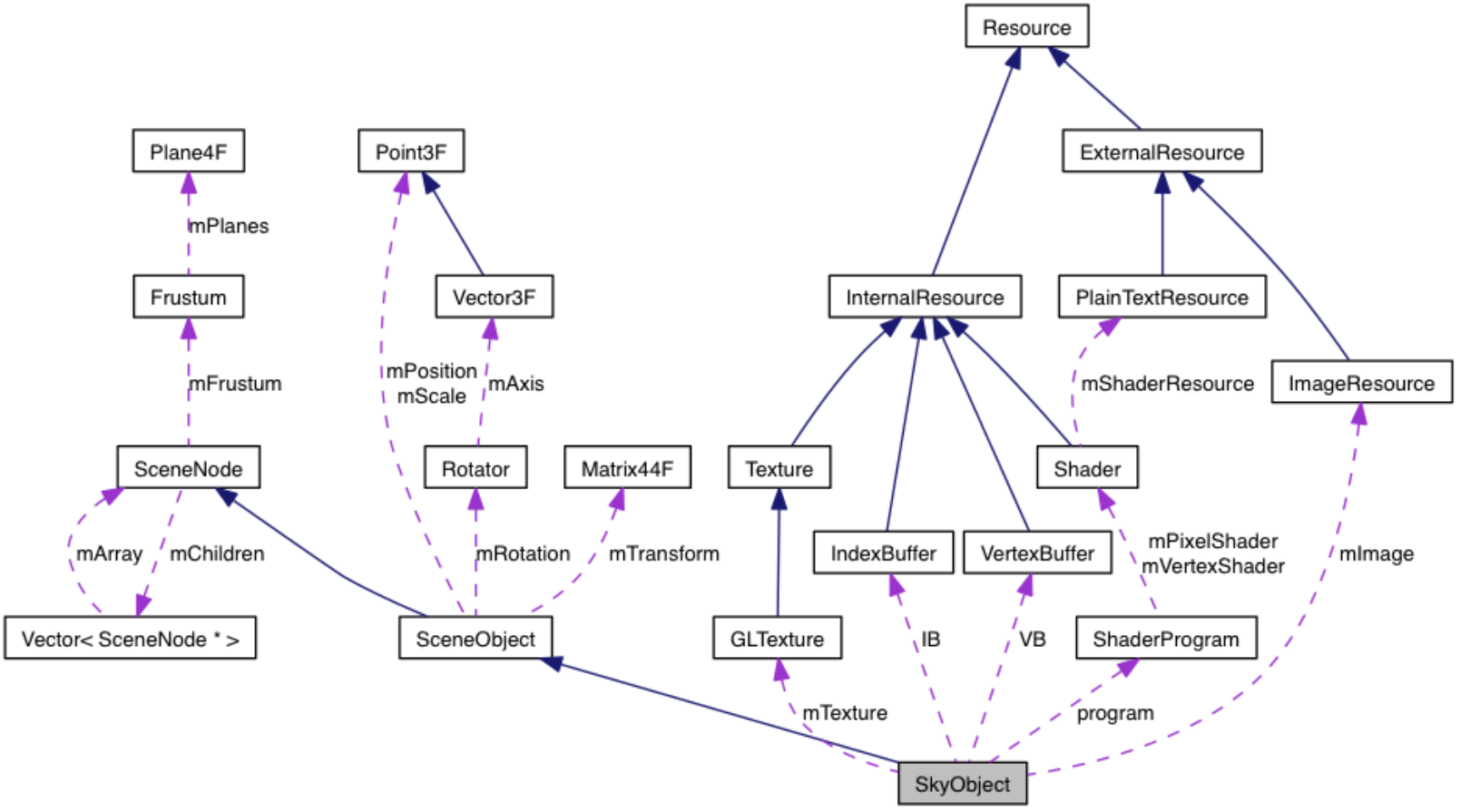

**Figure : UML sequence diagram for banking application**

From www-128.ibm.com/developerworks/rational/library/3101.html

# Translation Between Models and Code

- Graphical models typically *omit details* found in code
- Some graphical modeling tools can generate code "skeletons" from diagrams
- Some *reverse engineering* tools can generate graphical models from code
  - Code may be partial or complete
  - Obviates need to maintain separate diagrams
  - Facilitates updating of design documents

# Example: Doxygen-Generated Collaboration Diagram




taylorpetrick.files.wordpress.com/2011/08/skycollab.png

# Communicating the Design

- Communication between designers is critical to ensuring that
  - Design satisfies its requirements
  - Components can be integrated without difficulty
- Once a design is adopted, it must be communicated effectively to implementers, testers, maintainers, etc.

# The Design Document

- Provides *essential guidance* to implementers, testers, maintainers, etc.
- Should be *concise* and reasonably easy for developers to *understand*
- Consists of *prose* augmented with *diagrams*
- Should be *specific* enough to enable programmers to implement what the designer(s) intended

# Design Document cont.

- Identifies product or component it describes
- *References* relevant *requirements*
- Begins with high-level *overview*
- Identifies the *major components* and their *relationships* and *interactions*
- Contains or is supplemented with component *interface specifications*
- *Major design decisions* and their *rationales* should be made explicit

# Design Document cont.

- *Organization* of document should reflect the *structure* of the design
- Document should be *modular*
- Should be *cross-referenced* with requirements
- Should have *table of contents*, *index*, *glossary*, and *references* to relevant documents
- Should include dated *history of changes*
- Should be *inspected/reviewed* by team

# Design Presentations

- Designer(s) give slide presentation of design for
  - Reviewers
  - Management
  - Programmers
- *Graphical models* emphasized
- Major *design decisions* are explained and justified
- Questions answered
- *Issues* raised, recorded and addressed later