16 - Requirements Analysis (cont.) & Software Architecture

Rick Adrion

How shall we manage the interviews?

- Stakeholders to be interviewed
  - November 10
    - Department staff & associate chair
    - OIT
  - TBA
    - Registrar
    - Bursar
    - Yourselves
- Develop questionnaires
  - Pick the subset
    - Groups 1-4 “Records and enrollment”
    - Group 5 “ Bursar”
    - each group email me the subset

How shall we manage the interviews?

- Develop questionnaires
  - Pick the subset
  - Design questions

Administrative Staff

- Core Activities – Discuss these and identify other functions & interactions
  - Course Scheduling
    - Entering/deleting courses (and “course offerings”) in advance, as situations change (demand, instructor availability, etc.)
    - Notification of cancellations, schedule changes
    - Setting a semesters schedule (faculty assignments, room assignments, time assignments), access and display
  - Advising & Enrollment
    - Assigning PINs for access
    - Managing add/drop/exchange courses.
    - Enrollment minimums, maximums, wait lists, overrides
    - Supporting advising (providing records, audits, schedules, requirements, alternatives at the 5-colleges)
  - Grading and Grade Reporting, Transcripts and Certifications
    - Entering/changing/correcting grades
    - Maintaining course records (grade sheets, etc.)
    - Audits, accessing transcripts, maintaining student records
    - Managing withdrawals, leaves, student-year-abroad
    - Determining student status
    - Certifying eligibility for graduation

Administrative Staff

- Non-functional requirements – what do you expect based on your experience?
  - Ease of Use – User-friendliness, user interface, access controls
  - Reliability
  - Frequency of Use
- Communication – with whom do you interact?
  - Students
  - Faculty
  - Department Administration
  - Other Offices: Registrar, Classroom Scheduling Office, etc.
  - Peers
- Personal View
  - Good Experiences
  - Negative Experiences

How shall we manage the interviews?

- Develop questionnaires
  - Pick the subset
  - Design questions
  - Email them to me
- Plan and carry out interviews
  - 30-40 minutes/stakeholder-group
- Tasks:
  - Each group to email me "subset" by October 31
  - Each group to email me 2-3 categories with 2-5 "questions" for November 10 interview by November 3
  - Groups 1-4 email me 2-3 categories with 2-5 "questions" for Registrar interview by November 5
  - Group 5 email me 2-3 categories with 2-5 "questions" for Bursar interview by November 5

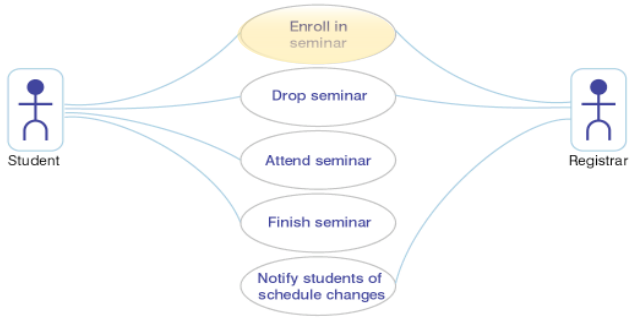
Dynamic Modeling with UML

- Diagrams for dynamic modeling
  - Interaction diagrams describe the dynamic behavior between objects
  - Statecharts describe the dynamic behavior of a single object
- Interaction diagrams
  - Sequence Diagram:
    - Dynamic behavior of a set of objects arranged in time sequence.
    - Good for real-time specifications and complex scenarios
  - Collaboration Diagram :
    - Shows the relationship among objects. Does not show time
- State Charts:
  - A state machine that describes the response of an object of a given class to the receipt of outside stimuli (Events).
- Activity Diagram:
  - Special type of statechart where all states are action states

Start with Flow of Events

- Get events from Use Case
- What is an Event?
  - something that happens at a point in time
- Relation of events to each other:
  - causally related: Before, after,
  - causally unrelated: concurrent
- An event sends information from one object to another
- Events can be grouped in event classes with a hierarchical structure.
- Event is often used in two ways:
  - Instance of an event class
  - Attribute of an event class

Example



Again, adapted from www.agilemodeling.com

Enroll in seminar Use-Case

Name: Enroll in Seminar
Identifier: UCxx
Description: Enroll an existing student in a seminar for which she is eligible.
Preconditions: The Student is registered at the University.
Postconditions: The Student will be enrolled in the course she wants if she is eligible and room is available.
Extends:
Includes:
Inherits From:
Basic Course of Action:
- see full Use-Case description

Enroll in seminar Use-Case

Basic Course of Action:

1. The student wants to enroll in a seminar.
2. The student inputs her name and student number into the system via "UI23 Security Login Screen."
3. The system verifies the student is eligible to enroll in seminars at the university, according to business rule "BR129 Determine Eligibility to Enroll."
4. The system displays "UI32 Seminar Selection Screen," which indicates the list of available seminars.
5. The student indicates the seminar in which she wants to enroll.
6. The system validates the student is eligible to enroll in the seminar, according to the business rule "BR130 Determine Student Eligibility to Enroll in a Seminar."
7. The system validates the seminar fits into the existing schedule of the student, according to the business rule "BR143 Validate Student Seminar Schedule."

Business Rule

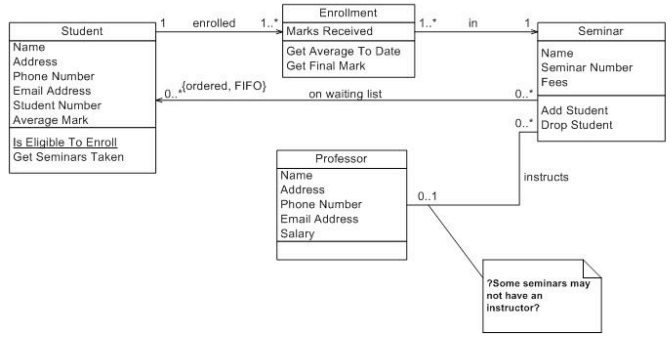
- Name
  - clear idea of topic, e.g., Determine Eligibility to Enroll
- Description
  - text, flow charts, UML activity diagrams, Object Constraint Language (OCL), Business Rules Markup Language (BRML)
- Example (optional)
- Source (optional)
- Related rules (optional)
- Revision history (optional)

Enroll in seminar Use-Case

Basic Course of Action:

1. The student wants to enroll in a seminar.
2. The student inputs her name and student number into the system via "UI23 Security Login Screen."
3. The system verifies the student is eligible to enroll in seminars at the university, according to business rule "BR129 Determine Eligibility to Enroll."
4. The system displays "UI32 Seminar Selection Screen," which indicates the list of available seminars.
5. The student indicates the seminar in which she wants to enroll.
6. The system validates the student is eligible to enroll in the seminar, according to the business rule "BR130 Determine Student Eligibility to Enroll in a Seminar."
7. The system validates the seminar fits into the existing schedule of the student, according to the business rule "BR143 Validate Student Seminar Schedule."

Class diagram



Specifying message sequences

- Useful to distinguish between
  - signals
    - asynchronous inter-object communication
    - often shown with "half-arrow notation"
  - Calls
    - synchronous inter-object communication control returns to caller (usually)

Sequence Diagram

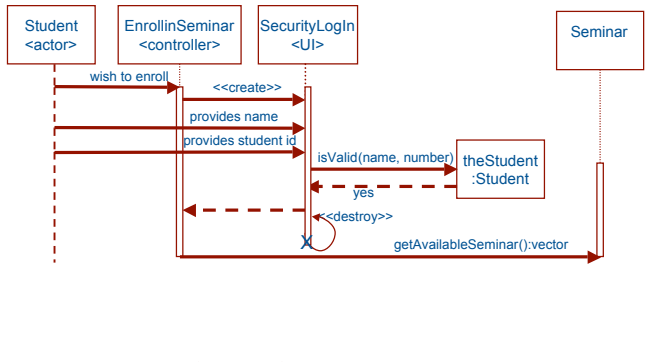
- From the flow of events in the use case or scenario proceed to the sequence diagram
- A sequence diagram is a graphical description of objects participating in a use case or scenario using a DAG notation
- Relation to object identification:
  - Many objects/classes have already been identified during object modeling
  - More objects are identified as a result of dynamic modeling
- Heuristic:
  - An event always has a sender and a receiver. Find them for each event => These are the objects participating in the use case

Enroll in seminar Use-Case

Basic Course of Action:

1. The student wants to enroll in a seminar.
2. The student inputs her name and student number into the system via "UI23 Security Login Screen." event
3. The system verifies the student is eligible to enroll in seminars at the university, according to business rule "BR129 Determine Eligibility to Enroll."
4. The system displays "UI32 Seminar Selection Screen," which indicates the list of available seminars. event
5. The student indicates the seminar in which she wants to enroll.
6. The system validates the student is eligible to enroll in the seminar, according to the business rule "BR130 Determine Student Eligibility to Enroll in a Seminar."
7. The system validates the seminar fits into the existing schedule of the student, according to the business rule "BR143 Validate Student Seminar Schedule."

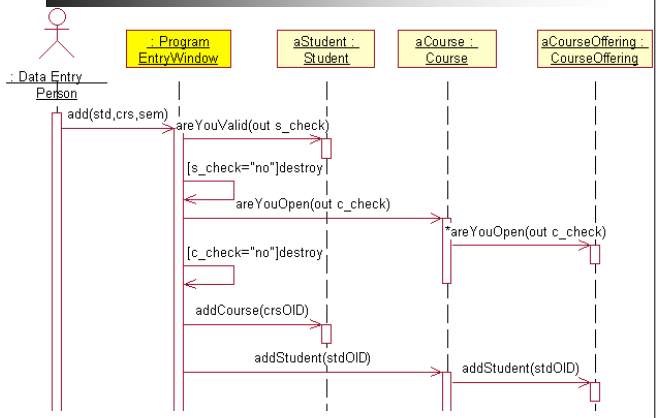
Sequence Diagram



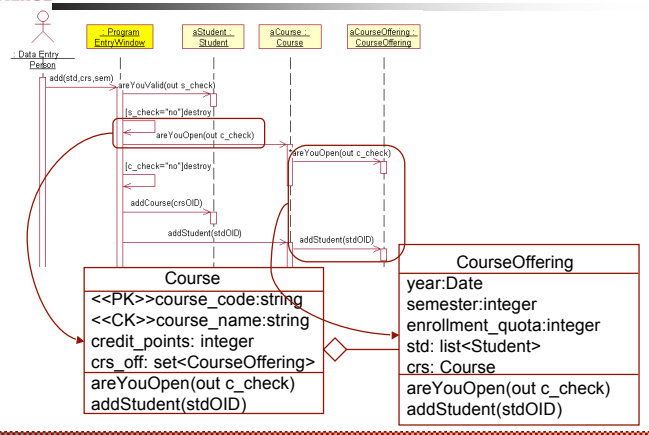
Defining Operations

- A public interface of a class consists of operations that offer services to entities external to the class
  - operations are best discovered from sequence diagrams, since every message must be serviced by an operation
- Other operations can be found using the CRUD (create, read, update, delete) paradigm; classes need to provide these services regardless of their domain specific functionality

Example 4.17 – Maciaszek



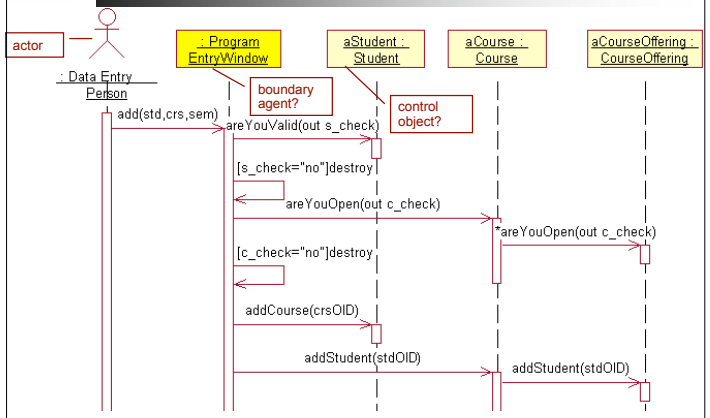
Example 4.17 – Maciaszek



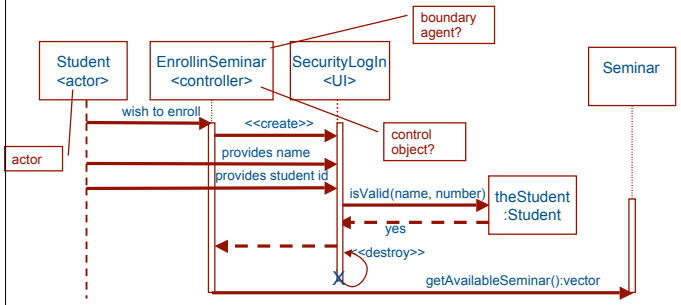
Heuristics for Sequence Diagrams

- Layout:
  - 1st column: Should correspond to the actor who initiated the use case
  - 2nd column: Should be a boundary object
  - 3rd column: Should be the control object that manages the rest of the use case
- Creation:
  - Control objects are created at the initiation of a use case
  - Boundary objects are created by control objects
- Access:
  - Entity objects are accessed by control and boundary objects,
  - Entity objects should never call boundary or control objects: This makes it easier to share entity objects across use cases and makes entity objects resilient against technology-induced changes in boundary objects.

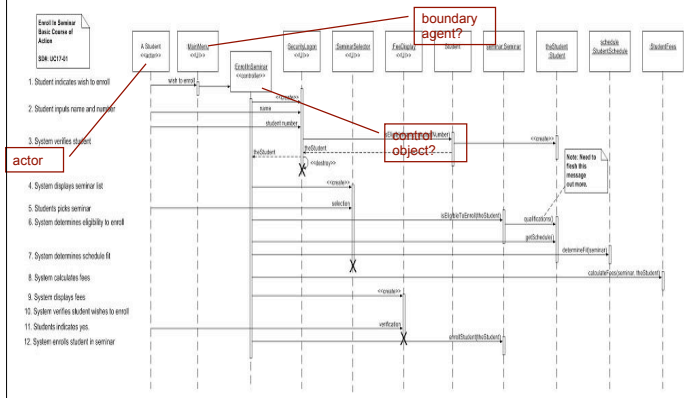
Is this a good sequence diagram?



Perhaps better ….



Perhaps even better ….



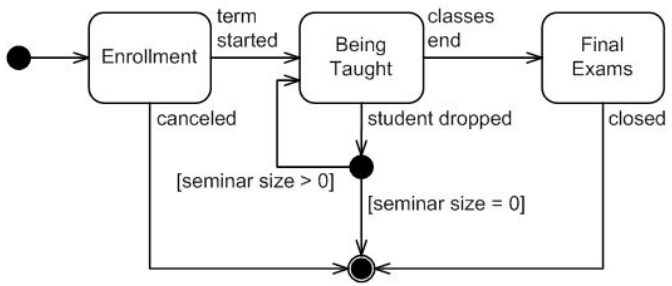
Modeling behavior

- An **object model** describes the possible patterns of **objects, attribute values and links** that can exist in a system.
- A **dynamic model** describes the possible patterns of **states, events and actions** that can exist in a system.
- Over time, the objects stimulate each other, resulting in a series of changes to their states.
- An individual stimulus from one object to another is called an event.
  - Events can be organized into classes.
  - Events can be error conditions as well as normal occurences.
  - Some events are only signals (OK),
  - Others have attributes associated with them

Statechart Diagrams

- Graph whose nodes are states and whose directed arcs are transitions labeled by event names.
- Distinguish between two types of operations:
  - Activity: Operation that takes time to complete
    - associated with states
  - Action: Instantaneous operation
    - associated with events
    - associated with states (reduces drawing complexity)
      - Entry, Exit, Internal Action
- A statechart diagram relates events and states for one class -> an object model with a set of objects has a **set** of state diagrams

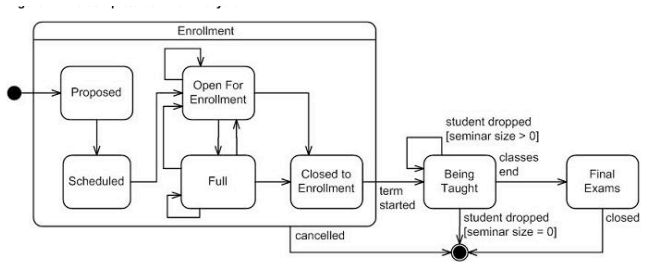
Example of a StateChart Diagram



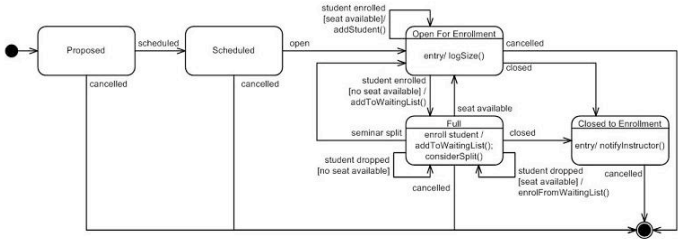
Nested State Diagram

- Structure
  - Activities in states are composite items denoting other lower-level state diagrams
  - A lower-level state diagram corresponds to a sequency of lower-level states and events that are invisible in the higher-level diagram.
  - Sets of substates in a nested state diagram denoting a superstate are enclosed by a large rounded box, also called contour.
- Nested state diagrams are useful:
  - As a solution to cope with the complexity in your design:
  - Abstraction
    - A state is actually more complex and leads to a finite state automaton itself. On the top level we don't model all the complex states.
  - Modularization
    - Each state diagram has up to 7+-2 states.
  - Apply the " Consist of" association!

Nested Statechart Diagram



Details of the Nested Statechart



Superstates

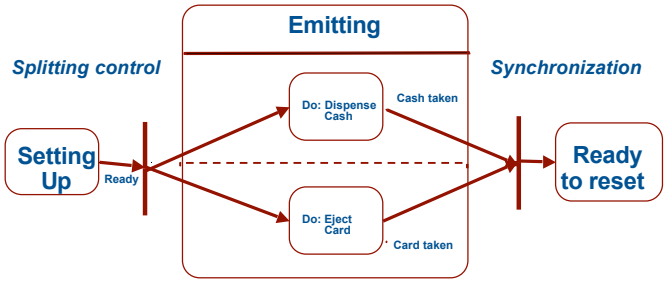
- Goal:
  - Avoid spaghetti models
  - Reduce the number of lines in a state diagram
- Transitions from other states to the superstate enter the first substate of the superstate.
- Transitions to other states from a superstate are inherited by all the substates (state inheritance)

Modeling Concurrency

Two types of concurrency

1. System concurrency
   - State of overall system as the aggregation of state diagrams, one for each object. Each state diagram is executing concurrently with the others.
2. Object concurrency
   - An object can be partitioned into subsets of states (attributes and links) such that each of them has its own subdiagram.
   - The state of the object consists of a set of states: one state from each subdiagram.
   - State diagrams are divided into subdiagrams by dotted lines.

Concurrency within an Object



Concurrency within an object

- Concurrency might be detected within a single object
  - overall decomposition of the system or initial object identification was too coarse grain?
- If there is concurrency ask
  - What objects are hidden in the currently modeled single objects?
  - May lead to new insights in the application or result in a better taxonomy or object model.
- In some cases, the object is inherently not further decomposable
  - ramifications during system design, e.g., mapping to multiple processors due to data parallelism
  - implementation, e.g., choice of programming language that supports lightweight threads instead of heavyweight processes

StateCharts vs Sequence Diagram

- State chart diagrams help to identify:
  - Changes to objects over time
- Sequence diagrams help to identify
  - The temporal relationship of between objects over time
  - Sequence of operations as a response to one ore more events

Practical Tips

- Construct dynamic models only for classes with significant dynamic behavior
  - couple of hundred objects not lead to 100's of dynamic models
  - define dynamic model for one that object that can be used to describe behavior of other ojects
- Consider only relevant attributes
  - Use abstraction if necessary
- Look at the granularity of the application when deciding on actions and activities
- Reduce notational clutter
  - Try to put actions into state boxes (look for identical actions on events leading to the same state)

Summary: Requirements Analysis

Functional Modeling
- What are the transformations?
  - Create scenarios and use case diagrams
    - Talk to client, observe, get historical records, do thought experiments

Object Modeling
- What is the structure of the system?
  - Create class diagrams
    - Identify objects. What are the associations between them? What is their multiplicity?
    - What are the attributes of the objects?
    - What operations are defined on the objects?

Dynamic Modeling
- 3. What is its control structure?
  - Create sequence diagrams
    - Identify senders and receivers
    - Show sequence of events exchanged between objects. Identify event dependencies and event concurrency.
  - Create state diagrams
    - Only for the dynamically interesting objects.

Software Architecture

Abstraction techniques in CS

- Programming Languages
  - machine language;
  - symbolic assemblers
  - macro processors
  - early high-level languages
    - Fortran
      - data types served primarily as cues for selecting the proper machine instructions
    - Algol and it successors
      - data types serve to state the programmer's intentions about how data should be used.
  - later high-level languages
    - separation of a module's specification from its implementation
    - introduction of abstract data types.

Abstraction techniques in CS

- ADT
  - the software structure (which included a representation packaged with its primitive operators)
  - specifications (mathematically expressed as abstract models or algebraic axioms)
  - language issues (modules, scope, user-defined types)
  - integrity of the result (invariants of data structures and protection from other manipulation)
  - rules for combining types (declarations)
  - information hiding (protection of properties not explicitly included in specifications)

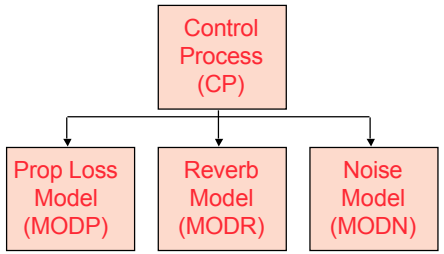
Software Architecture

- the theory of abstract data types
  - not the only way to organize a software system
- many have developed informally over time:
  - "Camelot is based on the **client-server model** and uses remote procedure calls both locally and remotely to provide communication among applications and servers."
  - "Abstraction layering and system decomposition provide the appearance of system uniformity to clients, yet allow Helix to accommodate a diversity of autonomous devices. The architecture encourages a **client-server model** for the structuring of applications."
  - "We have chosen a **distributed, object-oriented** approach to managing information."
  - "The easiest way to make the canonical sequential compiler into a concurrent compiler is to **pipeline** the execution of the compiler phases over a number of processors. . . . A more effective way [is to] split the source code into many segments, which are concurrently processed through the various phases of compilation [by **multiple compiler processes**] before a final, merging pass recombines the object code into a single program."

Other software architectures

- Open Systems Interconnection Reference Model (a layered network architecture)
- NIST/ECMA Reference Model (a generic software engineering environment architecture based on layered communication substrates)
- X Window System (a distributed windowed user interface architecture based on event triggering and callbacks)

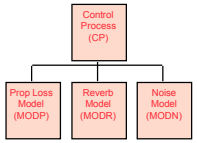
Example



- is this an architecture?

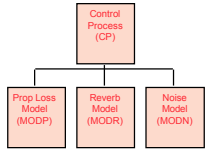
Example

- what is the nature of the components, and what is the significance of their separation?
  - do they run on separate processors?
  - do they run at separate times?
  - do the components consist of processes, programs, or both?
  - do the components represent ways in which the project labor will be divided, or do they convey a sense of runtime separation?
  - are they modules, objects, tasks, functions, processes, distributed programs, or something else?



Example

- what is the significance of the links?
  - do the links mean the components communicate with each other, control each other, send data to each other, use each other, invoke each other, synchronize with each other, or some combination of these or other relations?
- what is the significance of the layout?
  - why is CP on a separate (higher) level?
  - does it call the other three components, and are the others not allowed to call it?
  - was there simply not room enough to put all four components on the same row in the diagram?



Observations

- Designers freely use informal patterns/idioms
  - informal with imprecise semantics
  - diagrams + prose, but no rules
- Designers use system-level abstraction
  - overall organization (styles)
  - components and interactions
- Designers compose systems from subsystems
  - but, tend to think statically
  - select structure by default, rather than by design

Why Important?

- mutual communication.
  - software architecture represents a common high-level abstraction of the system that most, if not all, of the system's stakeholders can use as a basis for creating mutual understanding, forming consensus, and communicating with each other.
- early design decisions.
  - software architecture represents the embodiment of the earliest set of design decisions about a system, and these early bindings carry weight far out of proportion to their individual gravity with respect to the system's remaining development, its service in deployment, and its maintenance life.

Why Important?

- transferable abstraction of a system.
  - software architecture embodies a relatively small, intellectually graspable model for how the system is structured and how its components work together; this model is transferable across systems; in particular, it can be applied to other systems exhibiting similar requirements, and can promote large scale reuse.

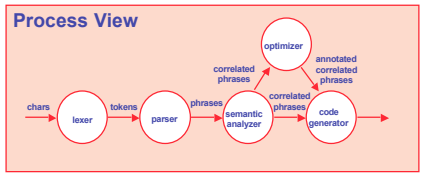
Embodies the Earliest Design Decisions

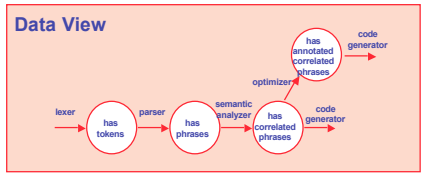
- architecture provides builders with constraints on implementation
- the architecture dictates organizational structure for development and maintenance projects
- an architecture permits or precludes the achievement of a system's targeted quality attributes
- it is possible to predict certain qualities about a system by studying its architecture
- architecture can be the basis for training
- an architecture helps to reason about and manage change

elements, form, rationale, views

architecture=

- elements
  - processing
  - data
  - connectors
- form
  - rules which constrain element placement
  - style/design
- rationale
  - selection of form
  - links to reqmnts & design
  - functional/non-functional attributes

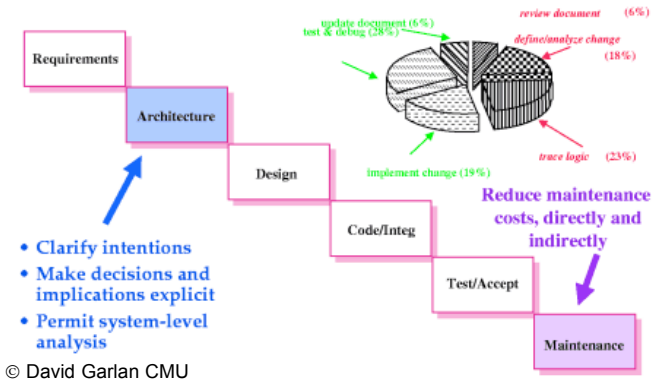




architectural styles/idioms

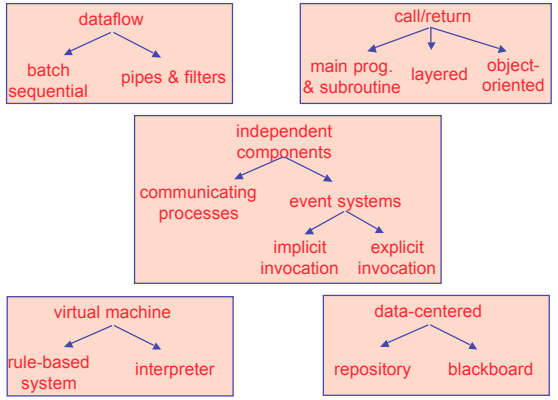
- architectural style =
  - Components: locus of computation
    - filters, databases, objects, clients, servers, ADTs
  - Connectors: mediate interactions of components
    - procedure call, pipes, event broadcast
  - Properties: specify info for construction & analysis
    - Signatures, pre/post conditions, RT specifications
- other
  - topology
  - underlying structural model?
  - underlying computational model?

Expected Benefits

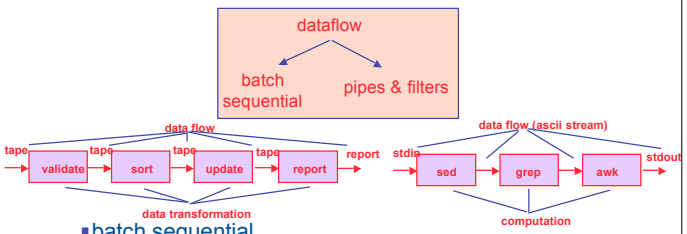


© David Garlan CMU

taxonomy



taxonomy:data flow



- batch sequential
  - independent programs, dataflow in large chunks, no parallelism
- pipes & filters
  - incremental, byte stream data flow, pipelined "parallelism", local context, no state persistence

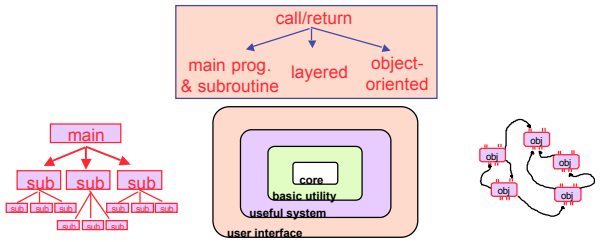
analysis

- pipes & filters (to some extent batch)
- problem decomposition
  - advantages: hierarchical decomposition of system function
  - disadvantages: "batch mentality",interactive apps?, design
- maintenance & reuse
  - advantages: extensibility, reuse, "black box" approach
  - disadvantages: lowest common denominator for data flow
- performance
  - advantages: pipelined concurrency
  - disadvantages: parsin/unparsing, queues, deadlock with limited buffers

Rules of thumb

- for dataflow
  - If your problem can be decomposed into sequential stages, consider batch sequential of pipeline architectures
    - If in addition each stage is incremental, so that later stages can begin before earlier stages complete, then consider a pipelined architecture
  - If your problem involves transformations on continuous streams of data (or on very long streams) consider a pipeline architecture
    - However, if your problem involves passing rich data representation, then avoid pipeline architectures restricted to ASCII
  - If your system involves controlling action, is embedded in a physical system, and is subject to unpredictable external perturbation so that preset algorithms go awry, consider a closed loop architecture

taxonomy: call/return



- main/sub
  - hierarchical decomposition, single thread of control, structure implicit, correctness depends on subordinates
- layered
  - hides lower layers/services higher layer, upper="virtual machines"/lower =hw, kernel, scoping
- object-oriented
  - encapsulation, inheritance, polymorphism

analysis

- layers
  - portability, modifiability, reuse
    - advantages: each layer is abstract machine, each layer interacts with ≤ 2 other layers, standard interfaces
  - performance, design
    - disadvantages: semantic feedback in UI, deep functionality, abstractions difficult, bridging layers
- object-oriented
  - portability, modifiability, reuse
    - advantages: decreased coupling, frameworks -> reuse
    - disadvantages: complex structure
  - performance, design
    - advantages: maps easily to "real world", inheritance, encapsulation
    - disadvantages: design harder, side effects, identity, inheitance difficult