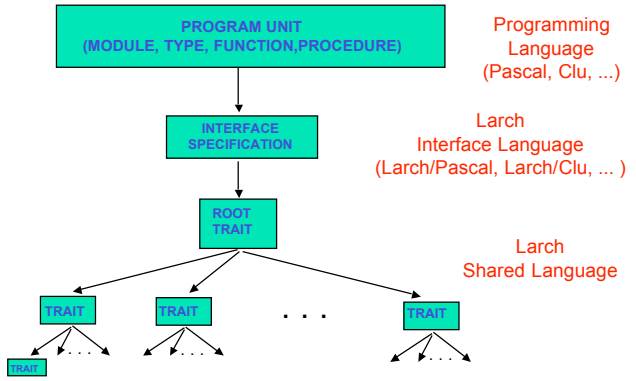
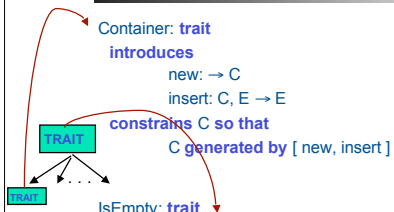
Today

- Finish Formal specifications
- 11 Introduction to Design

Larch

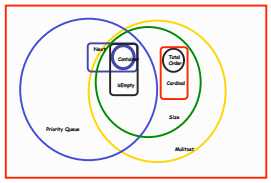


Traits



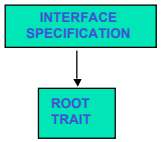
Container: **trait**
introduces
new: → C
insert: C, E → E
constrains C **so that**
C **generated by** [new, insert]

IsEmpty: **trait**
assumes Container
introduces
isEmpty: C → Bool
constrains isEmpty, new, insert
so that for all [*c* :C, *e* :E]
isEmpty(new) = true
isEmpty(insert(c,e)) = false
implies converts [isEmpty]



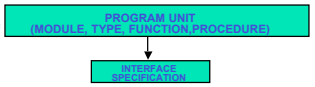
Larch/Pascal specification

type Bag exports bagInit, bagAdd, bagRemove, bagChoose
based on sort Mset from MultiSet with [integer for E]
procedure bagInit(var b:Bag)
modifies at most [b]
ensures bpost = { }
procedure bagAdd(var b:Bag; e; integer)
requires numElements(insert(b,e)) ≤ 100
modifies at most [b]
ensures bpost = insert(b,e)
procedure bagRemove(var b:Bag; e; integer)
modifies at most [b]
ensures bpost = delete(b,e)
procedure bagChoose(var b:Bag; e; integer): boolean
modifies at most [b]
ensures if ~ isEmpty (b)
then bagChoose & count (b, epost)>0
else ~ bagChoose & modifies nothing
End Bag



Pascal implementation of BagAdd

```
prodedure bagAdd(var B:Bag;e:integer);
  var i, lastEmpty: 1...MaxBagSize
  begin
    i:= 1;
    while ((i < MaxBagSize) and (b.elems[i]<>e)) do
      begin
        if b.counts[i] = 0 then LastEmpty:=i;
        i:= i+1;
      end;
    if b.elems[i] = e
      then b.counts[i]:= b.counts[i]+1;
      else begin
        if b.counts[i]=0 then LastEmpty:=i;
        b.elems[LastEmpty]:=e;
        b.counts[LastEmpty]:=1;
      end;
end[bagAdd];
```



Conclusions

- Interesting attempt to address:
  - readability/writability of formal specs
  - large, multi-lingual environment issues
- Relationship between shared and interface languages complex and unclear
- Relationship between interface and implementation languages not as strong as one would like
- "Software tool support needed" (syntax-directed editors, browsers, theorem-provers, etc.)

Current Status of Formal Specifications

- Strong theoretical foundation
- Some practical use, especially in Europe
- Current Languages trying to be more practical

How effective are these methods?

- Wing's study of the Library Problem
  - a small library database
  - transactions
    - checkout/return book
    - add/remove book
    - get a list of books
      - author
      - subject
      - borrower
    - get date/borrower for book
  - users
    - staff
    - borrowers
  - restrictions
    - availability
    - no book available & checked out
    - # books borrowed ≤max

Analysis

- Specification approaches
  - informal
  - AI
  - logic
  - executable/non-executable
- Comparisons
  - formality
  - life-cycle phase
  - operational vs. behavioral
  - modularity
  - readability
  - completeness
- Not considered
  - concurrency
  - reliability
  - fault-tolerance
  - security
- initialization
  - what's the initial state of the library?
- missing operations
  - need more transactions?
- error handling
  - what to do with errors?
  - checkout, return, add, remove, "type errors"
- missing constraints
  - more than one copy in library, checked out
- state
  - what to record, change?
- "non-functional" specification
  - human factors, liveness, time

Conclusions

- methods do not differ radically
- style
  - most use pre- and post-conditions for specifying behavior
  - algebraic & set-theoretic most common for specifying data (operational)
  - model-oriented (operational) most common approach
- formal specs can
  - identify diff in informal specs
  - handle simple, small problems
  - specify sequential functional behavior
- Challenges
  - scaling
  - non-functional behavior
  - combining techniques
  - tools
  - integrating specification into the lifecycle

11 Introduction to Design

- Reading
  - [GJM03] Fundamentals of Software Engineering by Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli, Second Edition, Prentice Hall; Chapter 4
  - [dlP72] Parnas David L., "On the criteria to be used in decomposing systems into modules," CACM, Dec., 1972
  - [dlP76] Parnas David L., "On the design and development of program families," IEEE Trans.SE., vol. SE-2, pp.1-9, Mar. 1976
  - [dlP79] Parnas, D.L. Designing software for ease of extension and contraction. In IEEE Trans. SE, Mar. 1979
  - Science of Design: Software-Intensive Systems A National Science Foundation Workshop Airlie Center, November 2-4, 2003 http://www.cs.virginia.edu/~sullivan/sdsis/workshop%202003.htm

What is design?

- Design
  - can refer to an activity or the result of the activity
  - acts as a bridge between requirements and the implementation of the software
- At a high level:
  - Requirements = "client's view"
    - What system is to do ⇒ **external view**
  - Design = "developer's view"
    - How the requirements are to be realized ⇒ **internal view**

What is design?

- Design gives a structure to the artifact
  - Decomposes system into parts, assigns responsibilities, ensures that parts fit together to achieve a global goal
    - e.g., a requirements specification document must be *designed*
  - The structure must be easy to understand and evolve
- Design is iterative & continuous
  - High-Level Design
    - Components & Connections
  - Low-Level Design
    - Representation & Algorithms
  - Very-Low-Level Design
    - Naming, Constructs, etc.

Design Decisions

- many (unbounded?) number of designs that satisfy (some?) of the requirements
  - thousands of decisions may go into a single page of code
- how to choose among these alternatives?
  - criteria: identify, reject, select, evaluate
  - strategy: **manage complexity, accommodate change, consider product families**

Managing complexity

- "Divide and Conquer" & "Separation of Concerns"
  - Need to decompose large systems in order to build them
  - But, composition may be as or more important
    - "Divide and conquer. Separate your concerns. Yes. But sometimes the conquered tribes need to be reunited under the conquering ruler, and the separated concerns must be combined to serve a single purpose …" Jackson, 1995
- Decomposition techniques are different for software than those used in physical systems
  - Fewer constraints are imposed by the material
  - Does the "Shanley Principle" (one part can perform multiple functions) hold?

Decomposition

- Benefits
  - Decrease size of tasks
  - Support independent testing and analysis
  - Separate work assignments
  - Ease understanding
- How do we select a decomposition?
  - We determine the desired criteria & select a decomposition (design) that will achieve those criteria
  - But it's hard to
    - Determine the desired criteria with precision, resolve tradeoffs
    - Determine if a design satisfies given criteria or find a better one that (better) satisfies (more) criteria
  - It may easy to build something designed pretty much like the last one
    - benefits: understandability, properties of the pieces, etc.

Decomposition Issues

- Structure
  - current design approaches focus on structure
    - What are the components and how are they put together?
    - Behavior is important, but largely indirectly
  - however, organizations and individuals often buy into a particular approach or methodology
    - "Beware a methodologist who is more interested in his methodology than in your problem." —M. Jackson
- Conceptual integrity
  - a critical design criterion?
    - "It is better to have a system omit certain anomalous features and improvements, but to reflect one set of design ideas, than to have one that contains many good but independent and uncoordinated ideas." —Brooks, MMM
  - makes it far easier to decide what is easy and reasonable to do as opposed to what is hard and less reasonable to do
    - not always what management wants to hear

Accommodating change

"…accept the fact of change as a way of life, rather than an untoward and annoying exception." —Brooks, 1974

"Software that does not change becomes useless over time." —Belady and Lehman

- Internet time makes the need to accommodate change even more apparent
- It is generally believed that to accommodate change one must anticipate possible changes
- By anticipating (and perhaps prioritizing) changes, one defines additional criteria for guiding the design activity
- However, it is not possible to anticipate all changes

Counterpoint

- Extreme Programming argues otherwise.
  - in essence it asserts that we are unable to effectively predict change
  - instead one should at every point use the simplest possible design for a software system
  - once changes are needed, one should restructure the design to meet the needs
  - questions conventional wisdom but it is still quite early and the outcome is unclear

Why design for change?

- Change of underlying abstract machine
  - new release of operating system
  - new optimizing compiler
  - new version of DBMS
- Change of peripheral devices
- Change of "social" environment
  - new tax regime
  - € versus former national currencies in EU
- Change due to development process (transform prototype into product)
- Change in algorithms, data representation
  - inefficient sorting algorithm ⇒ a more efficient one
  - binary tree ⇒ threaded tree
    - ~17% of maintenance costs attributed to data representation changes (Lientz and Swanson, 1980)

Product families

- Different versions of the same system
  - e.g. a family of mobile phones
    - members of the family may differ in network standards, end-user interaction languages, …
  - e.g. a facility reservation system
    - for hotels: reserve rooms, restaurant, conference space, …, equipment (video beamers, overhead projectors, …)
    - for a university
      - many functionalities are similar, some are different (e.g., facilities may be free of charge or not)

Product families

- Design goal for family
  - Design the whole family as one system, not each individual member of the family separately
- Sequential completion: the wrong way
  - Design first member of product family & modify existing software to get next member products
- How to do better
  - Anticipate definition of all family members
  - Identify what is common to all family members, delay decisions that differentiate among different members

Properties of design

- Cohesion
- Coupling
- Complexity
- Correctness
- Correspondence

Cohesion & Coupling

- Cohesion: The reason that elements are found together in a module
  - Ex: coincidental, temporal, functional, …
  - The details aren't critical, but the intent is useful
  - During maintenance, one of the major structural degradations is in cohesion
- Coupling: Strength of interconnection between modules
  - Hierarchies are touted as a wonderful coupling structure, limiting interconnections
  - But don't forget about composition, which requires some kind of coupling
- It's easy to...
  - ...reduce coupling by calling a system a single module
  - …increase cohesion by calling a system a single module

Coupling

- Coupling also degrades over time
  - "I just need one function from that module…"
- Unnecessary coupling hurts
  - Propagates effects of changes more widely
  - Harder to understand interfaces (interactions)
  - Harder to understand the design
  - Complicates managerial tasks
  - Complicates or precludes reuse

Coupling

- No satisfactory measure of coupling
  - Either across modules or across a system
  - Cruickshank and Gaffney Coupling metric

$$Coupling = \frac{\sum_{i=1}^{n} Z_i}{n}$$

$$where: \quad Z_i = \frac{\sum_{j=1}^{m} M_j}{m}$$

M_j = sum of the number of input and output items shared between components i & j

Z_i = average number of input and output items shared over m components with component i

n = number of components in the software product

Complexity

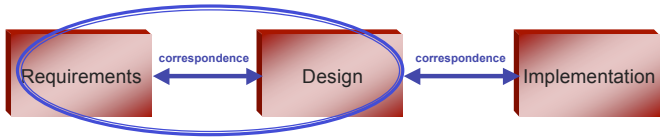
- simpler designs are better, all else being equal
- no useful measures of design/program complexity exist
  - Although there are dozens of such measures
  - LOC seems to be the most reliable predictor when single domains are considered, e.g.
    - data processing
    - numerical processing
    - symbolic processing e.g., compilers
    - concurrent/distributed systems e.g., operating systems

Correctness

- Very difficult (we'll come back to this briefly later in the course)
- Even if you "prove" modules are correct, how do you prove
  - Composition of the modules within and outside the system to be designed
  - System software that must interpret/compile and run the composed modules
  - Hardware on which the complied modules run, etc.
- Leveson and others have shown clearly that a system can fail even when each of the pieces work properly

Correspondence

- "Problem-program mapping"
- The way in which the design is associated with the requirements
- The idea is that the simpler the mapping, the easier it will be to accommodate change in the design when the requirements change
- See M. Jackson: problem frames



Functional decomposition

- Divide-and-conquer based on functions

```
input;
compute;
output
```

  - Then proceed to decompose compute
- This is stepwise refinement (Wirth, 1971)
  - There is an enormous body of work in this area, including many formal calculi to support the approach
  - Closely related to proving programs correct
  - More effective in the face of stable requirements

Properties

- What are the desirable properties of a modular structure?
- Almost all the literature focuses on logical structures in design, but physical structure plays a big role in practice
  - Sharing
  - Separating work assignments
  - Degradation over time
- Why so little attention paid to this?

Module

- To what degree do you consider your systems
  - as having modules?
  - as consisting of a set of files?
- This is a question of physical vs. logical structure of programs
  - In some languages/environments, they are one and the same
  - Ex: Smalltalk-80
- How to define the structure of a modular system?
  - A **module** is a well-defined **component** of a software system
  - A **module** is part of a system that **provides a set of services** to other modules
    - where services are computational elements that other modules may use

Modules and relations

- Let S be a set of modules
  $S = \{M_1, M_2, \ldots, M_n\}$
- A binary relation* $\mathcal{R}$ on S is a subset of
  $\mathcal{R} \subseteq S \times S$
- If M_i and M_j are in S, $\langle M_i, M_j \rangle \in \mathcal{R}$ can be written as
  $M_i \mathcal{R} M_j$
- Transitive closure $\mathcal{R}^+$ of $\mathcal{R}$ $M_i \mathcal{R}^+ M_j$ iff
  $M_i \mathcal{R} M_j$ or $\exists M_k$ in S s.t. $M_i \mathcal{R} M_k$ and $M_k \mathcal{R}^+ M_j$
- $\mathcal{R}$ is a hierarchy iff there are no two elements M_i, M_j s.t.
  $M_i \mathcal{R} M_j \cap M_j \mathcal{R} M_i$
- Relations can be represented as graphs; a hierarchy is a DAG (directed acyclic graph)

*we assume our relations to be irreflexive

The `uses` relation

- `A uses B`; examples
  - `A` requires the correct operation of `B`
  - `A` can access the services exported by `B` through its interface
  - `A` depends on `B` to provide its services
    - example: `A` calls a routine exported by `B`
  - `A` is a client of `B`; `B` is a server
  - the correctness of `A` depends on the presence of a correct version of `B`
    - requires specification and implementation of `A` and the specification of `B`
- Criteria for `uses(A,B)`
  - `A` is essentially simpler because it uses `B`
  - `B` is not substantially more complex because it does not use `A`
  - There is a useful subset containing `B` but not `A`
  - There is no useful subset containing `A` but not `B`

The `uses` relation

- `uses` should be a hierarchy
  - Hierarchy makes software easier to understand
  - Proceed from leaf nodes (who do not use others) upwards
  - They make software easier to build
  - They make software easier to test
- A non-hierarchical uses relation makes it difficult to produce useful subsets of a system -- Parnas
  - It also makes testing difficult
  - (What about upcalls?)
  - So, it is important to design the `uses` relation
- Can `uses` be mechanically computed?

`uses` vs. `invokes`

- These relations often but do not always coincide
  - Invocation without use: name service with cached hints

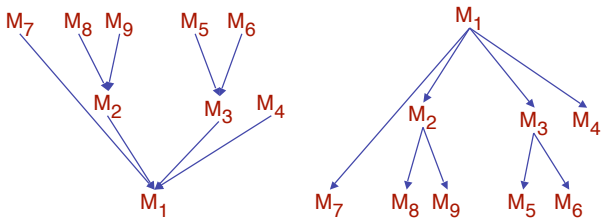
```
ipAddr := cache(hostName);
if wrong(ipAddr,hostName) then
   ipAddr := lookup(hostName)
endif
```

  - Use without invocation: examples?

The `is_component_of` Relation

- Used to describe a higher level module as constituted by a number of lower level modules
- `A is_component_of B`
  - `B` consists of several modules, of which one is `A`
  - `B` **`comprises`** `A`
  - **If** $M_{S,i}=\{M_k | M_k \in S \wedge M_k$ `is_component_of` $M_i\}$ **then** we say that $M_{S,i}$ **`implements`** M_i
- Careful recording of (hierarchical) `uses` and `is_component_of` relations supports design of program families

A graphical view of is_component_of



(is_component_of) (comprises)

They are a hierarchy

Hierarchy

- Organizes the modular structure through levels of abstraction
- Each level defines an abstract (virtual) machine for the next level
  - level can be defined precisely
    - M_i has level 0 if no M_j exists s.t. $M_i \mathcal{R} M_j$
    - let k be the maximum level of all nodes M_j s.t. $M_i \mathcal{R} M_j$ … then M_i has level k+1

Information hiding

- Information hiding is perhaps the most important intellectual tool developed to support software design [Parnas 1972]
  - Makes the anticipation of change a centerpiece in decomposition into modules
- Provides the fundamental motivation for abstract data type (ADT) languages
  - And thus a key idea in the OO world, too
- The conceptual basis is key

Basics of information hiding

- Modularize based on anticipated change
- Separate interfaces from implementations
  - Implementations capture decisions likely to change
  - Interfaces capture decisions unlikely to change
  - Clients know only interface, not implementation
  - Implementations know only interface, not clients
- Modules are also work assignments

Anticipated changes

- most common anticipated change is "change of representation"
  - a key notion behind abstract data types
  - e.g., Cartesian vs. polar coordinates; stacks as linked lists vs. arrays; packed vs. unpacked strings
- do we change representations less frequently today?
  - more knowledge about data structure design
  - memory is much less expensive
- so, think twice about anticipating that representations will change
  - important, since we can't simultaneously anticipate all changes

Algorithm changes

- almost always part and parcel of ADT-based decompositions
- monolithic to incremental algorithms
- improvements in algorithms
- information hiding isn't only using ADTs

- Other changes?

Notkin's IH "Central Premises"

1. can effectively anticipate changes
   - essentially no research and we have no disciplined ways to anticipate changes
   - but, unanticipated changes require changes to interfaces or (more commonly) simultaneous changes to multiple modules
2. changing an implementation is the best change, since it's isolated
   - changing a local implementation may not be easy

Notkin's IH "Central Premises"

3. semantics of a module must remain unchanged when implementations are replaced
   - what captures the semantics of the module? signature of the interface? performance? what else?
4. one implementation can satisfy multiple clients
   - clients of the same interface that need different implementations is counter to the principle of information hiding
5. information hiding can be recursively applied
   - Is this true? If not, what are the consequences?

Information Hiding and OO

- Are these the same? Not really
  - OO classes are chosen based on the domain of the problem (in most OO analysis approaches)
  - Not necessarily based on change
- But they are obviously related (separating interface from implementation, e.g.)
- What is the relationship between sub- and super-classes?

Interface design

- Interface should not reveal what we expect may change later
  - It should not reveal unnecessary details
  - Interface acts as a firewall preventing access to hidden parts
- Prototyping
  - Once an interface is defined, implementation can be done
    - first quickly but inefficiently
    - then progressively turned into the final version
  - Initial version acts as a prototype that evolves into the final product

Interface vs. implementation

- To understand the nature of `uses`, we need to know what a used module exports through its interface
  - The client imports the resources that are exported by its servers
  - Modules implement the exported resources
  - Implementation is hidden to clients
- Clear distinction between interface and implementation is a key design principle
  - Supports separation of concerns
    - clients care about resources exported from servers
    - servers care about implementation
  - Interface acts as a contract between a module and its clients

Layering [Parnas 79]

- Focus on information hiding modules isn't enough
- One may also consider abstract machines
  - In support of program families
    - Systems that have "so much in common that it pays to study their common aspects before looking at the aspects that differentiate them"
- Still focusing on anticipated change

Language support

- We have lots of language support for information hiding modules
  - C++ classes, Ada packages, etc.
- We have essentially no language support for layering
  - Operating systems provide support, primarily for reasons of protection, not abstraction
  - Big performance cost to pay for "just" abstraction

Categories of modules

- Functional modules
  - traditional form of modularization
  - provide a procedural abstraction
  - encapsulate an algorithm
- Libraries
  - a group of related procedural abstractions
    - e.g., mathematical libraries
      - implemented by routines of programming languages
- Common pools of data
  - data shared by different modules
    - e.g., configuration constants
      - the COMMON FORTRAN construct

Abstract Modules

- Abstract objects
  - Objects manipulated via interface functions
  - Data structure hidden to clients
- Abstract data types
  - Many instances of abstract objects may be generated

Abstract objects

- Examples
  - calculator of expressions expressed in Polish postfix form: a*(b+c) ◊ abc+*
  - a stack where the values of operands are shifted until an operator (assume only binary operators) is encountered in the expression

    Interface of the abstract object STACK
    exports
    procedure PUSH (VAL: in integer);
    procedure POP_2 (VAL1, VAL2: out
  - How does the design anticipate change in type of expressions to be evaluated?
    - e.g., it does not adapt to unary operators

Abstract data types (ADTs)

- Example: stack ADT

```
module  STACK_HANDLER
exports
        type  STACK = ?;
        This is an abstract data -type module; the data structure
        is a secret hidden in the implementation part.
        procedure   PUSH (S:  in out  STACK ; VAL:  in  integer);
        procedure   POP (S: in out  STACK ; VAL:  out  integer);
        function  EMPTY (S: in  STACK) : BOOLEAN;
        :
end  STACK_HANDLER
```

indicates that details of the data structure are hidden to clients

- ADTs correspond to Java and C++ classes & may also be implemented by Ada private types and Modula-2 opaque types

Abstract data types (ADTs)

- Aother example: simulation of a gas station

```
module FIFO_CARS
uses CARS
exports
        type QUEUE : ?;
        procedure ENQUEUE (Q: in out QUEUE ; C: in CARS);
        procedure DEQUEUE (Q: in out QUEUE ; C: out CARS);
        function IS_EMPTY (Q: in QUEUE) : BOOLEAN;
        function LENGTH (Q: in QUEUE) : NATURAL;
        procedure MERGE (Q1, Q2 : in QUEUE ; Q : out QUEUE);
        This is an abstract data-type module representing
        queues of cars, handled in a strict FIFO way;
        queues are not assignable or checkable for equality,
        since ":=" and "=" are not exported.
        …
end FIFO_CARS
```


Generic modules

- parametric with respect to a type

```
generic module GENERIC_STACK_2
. . .
exports
procedure PUSH (VAL : in T);
procedure POP_2 (VAL1, VAL2 : out T);
…
end GENERIC_STACK_2
```

- specify that a type and also an operation must be provided parameters

```
generic module M (T) with OP(T)
uses ...
        ...
end M
```

- instantiation syntax:

```
module INTEGER_STACK_2 is GENERIC_STACK_2 (INTEGER)
module M_A_TYPE is M(A_TYPE)  PROC(M_A_TYPE)
```

More on genericit

Software Architecture

- architecture of a system describes its gross structure
- illuminates the top level design decisions
  - how the system is composed of interacting parts
  - the main pathways of interaction
  - the key properties of the parts
- allows high-level analysis and critical appraisal

Roles of Software Architecture

- a bridge between requirements and implementation
  - an abstract description of a system,
  - exposes certain properties, while hiding others.
- useful for:
  - Understanding
  - Reuse
  - Construction
  - Evolution
  - Analysis
  - Management

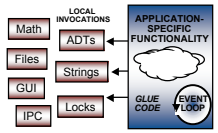
Roles of Software Architecture

- Understanding:
  - simplifies the understanding of large systems using an abstraction
  - constraints on system design
  - rationale
- Construction
  - a partial blueprint for development: components and dependencies
- Evolution
  - dimensions along which a system is expected to evolve
  - "load-bearing walls" -> ramifications of changes, cost estimation
  - separate concerns about the functionality of a component from the ways in which that component is connected to (interacts with) other components
- Analysis
  - consistency checking
  - conformance
    - to constraints
    - to quality attributes
  - dependence analysis
  - domain-specific analyses for architectural styles
- Reuse
  - reuse of large components and frameworks
- Management
  - leads to a much clearer understanding of requirements, implementation strategies, and potential risks

Software Architectures

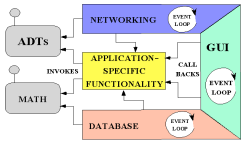
- Architectural taxonomy ("boxology")
- Architectural patterns & idioms
- Design patterns & idioms
- Reuse
  - Class libraries
  - Components
  - Frameworks
  - Middleware

Frameworks & Class Libraries



Class Library Architecture

- A class is a unit of abstraction & implementation in an OO programming language

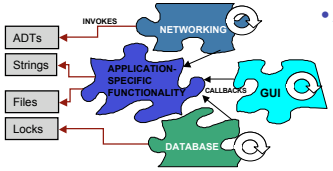


Framework Architecture

- A framework is an integrated set of abstract classes that can be customized for instances of a family of applications

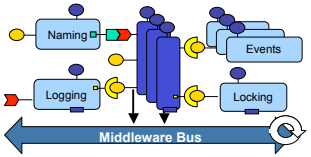
Adapted from Douglas C. Schmidt, "Patterns, Frameworks, & Middleware: Their Synergistic Relationships"

Frameworks & Components



Framework Architecture

- A framework is an integrated set of abstract classes that can be customized for instances of a family of applications



Component Architecture

- A component is an encapsulation unit with one or more interfaces that provide clients with access to its services

Adapted from Douglas C. Schmidt, "Patterns, Frameworks, & Middleware: Their Synergistic Relationships"

Comparison

Class Libraries	Frameworks	Components
Micro-level	Meso-level	Macro-level
Stand-alone language entities	"Semi-complete" applications	Stand-alone composition entities
Domain-independent	Domain-specific	Domain-specific or Domain-independent
Borrow caller's thread	Inversion of control	Borrow caller's thread

Taxonomy of Patterns & Idioms

Type	Description	Examples
Idioms	Restricted to a particular language, system, or tool	Scoped locking
Design patterns	Capture the static & dynamic roles & relationships in solutions that occur repeatedly	Active Object, Bridge, Proxy, Wrapper Façade, & Visitor
Architectural patterns	Express a fundamental structural organization for software systems that provide a set of predefined subsystems, specify their relationships, & include the rules and guidelines for organizing the relationships between them	Half-Sync/Half-Async, Layers, Proactor, Publisher-Subscriber, & Reactor
Optimization principle patterns	Document rules for avoiding common design & implementation mistakes that degrade performance	Optimize for common case, pass information between layers