What is Design?

- Design creates a representation or model of the software
- but unlike the analysis model, the design model provides detail about:
  - » *software architecture*,
  - » *data structures*,
  - » *interfaces*, and
  - » *components* that are necessary to implement the system.
- Why is it important?
  - ✓ Because the model can be assessed for quality and improved before code is generated, tests are conducted, and more users are involved.

Characteristics of good design

According to McGlaughlin(1991):

- Must ***implement*** all of the ***explicit*** requirements contained in the analysis model, and it must ***accommodate*** all of the ***implicit requirements*** desired by the customer.
- Must be a ***readable***, ***understandable*** guide for those who ***generate code*** and for those who ***test*** and subsequently ***support*** the software.
- Should provide a ***complete picture*** of the software, addressing the data, functional, and behavioral domains from an implementation perspective.

Design Principles

According Davis(1995):

- The design process should not suffer from ‘tunnel vision.’
- The design should be traceable to the analysis model.
- The design should not reinvent the wheel.
- The design should “minimize the intellectual distance” [DAV95] between the software and the problem as it exists in the real world.
- The design should exhibit uniformity and integration.

Design Principles

- The design process should not suffer from ‘tunnel vision.’

- The design should not reinvent the wheel.
  - Reinventing the wheel is a phrase that means to duplicate a basic method that has already previously been created or optimized by others.

Design Principles

- The design should be structured to accommodate change.
- The design should be structured to degrade gently, even when irregular data, events, or operating conditions are encountered.
- Design is not coding, coding is not design.
- The design should be assessed for quality as it is being created, not after the fact.
- The design should be reviewed to minimize conceptual (semantic) errors.

Generic Task Set

- **Design data structures, or data objects and their attributes as appropriate to the information domain model**
- **Using the analysis model decide on the architectural style appropriate for the software**
- **Partition the analysis model into design subsystems and allocate these subsystem within the architecture:**
  - Ensure that each subsystem is functionally cohesive
  - Design subsystem interfaces
  - Allocate analysis classes or functions to each subsystem

Generic Task Set

- **Create a set of design classes or components:**
  - Translate each analysis class description into a design class
  - Check each design class against quality criteria
  - Consider inheritance issues
  - Define methods and messages for each design class
  - Evaluate and select design patterns for each design class or a subsystem
  - Review the design classes and revise if needed
- **Design interfaces to external systems or devices**

Generic Task Set

- **Design the user interface:**
  - Review results of task analysis
  - Specify action sequences based on user scenarios
  - Create the behavioural model of the interface
  - Define interface objects and control mechanisms
  - Review the interface design and revise, if needed
- **Conduct component-level design:**
  - Specify all algorithms at a relatively low level of abstraction
  - Refine the interface of each component
  - Define component-level data structures
  - Review each component and correct any errors found
- **Develop a deployment model**

Design Concepts

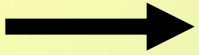
- Fundamental design concepts that span both traditional and OO software development include:

1) **Abstraction**
2) **Architecture**
3) **Patterns**
4) **Separation of concerns**
5) **Modularity**
6) **Information hiding**
7) **Functional Independence**
8) **Refinement**
9) **Aspects**
10) **Refactoring**
11) **OO Design concepts**
12) **Design classes**

Abstraction

- Designers should work to derive both procedural and data abstractions that serve the problem.
- Procedural abstraction – sequence of instructions that have a specific and limited function
- Data abstractions – a named collection of data that describes a data object

Data Abstraction



door

manufacturer
model number
type
swing direction
inserts
lights
  type
  number
weight
opening mechanism

implemented as a data structure

Procedural Abstraction

open

details of enter algorithm

implemented with a "knowledge" of the object that is associated with enter

Architecture

- Is concerned on:
  - describing the fundamental organization of the system,
  - identifying its various components and their relationships to each other, and
  - the environment in order to meet the system's quality goals.
- Also, describe the overall structure of the software :
  - organization of program modules,
  - the manner in which these modules interact, and
  - the structure of data used by the components.

Patterns

A pattern is a description of a common problem and a likely solution, based on experience with similar situations

http://www.mitchellsoftwareengineering.com/IntroToDesignPatterns.pdf

http://sourcemaking.com/design_patterns

http://www.voelter.de/data/pub/MDDPatterns.pdf

Separation of Concerns

- Actually, it is a rule of thumb to define how modules should be separated to each other:
  - different or unrelated concerns should be restricted to different modules.
- Suggests that any complex problem can be easily handled if it is sub-divided into pieces that can be solved independently.
- Why? So that a problem takes less time and effort to solve.
- Is **manifested in other design concepts**: modularity, aspects, functional independence and refinement.

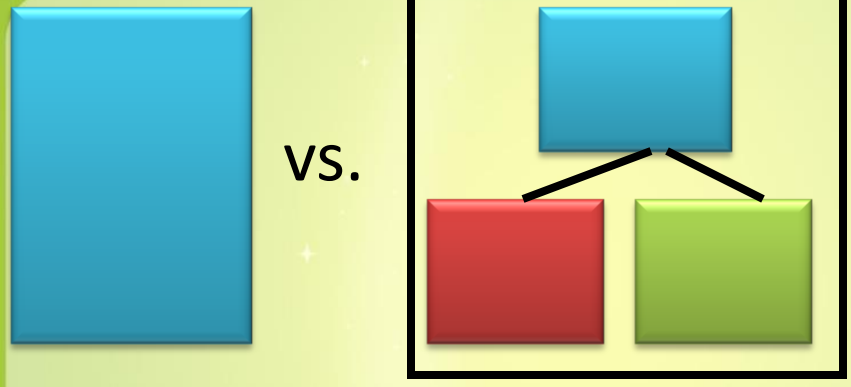
Modularity

- What is a module?
  - A lexically contiguous sequence of program statements, bounded by boundary elements, with an aggregate identifier
  - Examples of boundary elements are begin…end pairs or {…} pairs
  - Procedures, functions in classical paradigm are modules. In object-oriented paradigm an object is a module and so is the method within an object

Modularity

- Modularize a design:
  - To ease the planning for implementation (coding,
  - To define and deliver software increments,
  - To easily accommodate changes,
  - To efficiently test and debug program, and
  - To conduct long-term maintenance without serious side effects

Modularity Example



Without Module

```
#include <stdio.h>
main()
{ …
   printf("This program
   draws a rectangle");
   …
}
```

With Module

```
#include <stdio.h>
void print_menu(void);
main()
{ …
   print_menu();
   …
} /* end main */

void print_menu(void)
{
   printf("This program
   draws a rectangle");
} /* end function */
```

Information Hiding

- Suggests that modules should be specified and designed so that information (data structures and algorithm) contain within a module is inaccessible to other modules that have no need for such information.
- Implies that effective modularity can be achieved by defining a set of independent modules that communicate with one another only necessary information.

Information Hiding

- Serves as an effective criterion for dividing any piece of equipment, software or hardware, into modules of functionality.
- Provides flexibility.
  - This flexibility allows a programmer to modify functionality of a computer program during normal evolution as the computer program is changed to better fit the needs of users.
  - When a computer program is well designed decomposing the source code solution into modules using the principle of information hiding, evolutionary changes are much easier because the changes typically are local rather than global changes.

Information Hiding : Example

- Suppose we have a `Time` class that counts the time of day:

```
class Time
{
public:
  void Display();
private:
  int ticks;
};
```

- The `Display()` member function prints the current time onscreen. This member function is accessible to all. It's therefore declared public.
- By contrast, the data member `ticks` is declared private. Therefore, external users can't access it.

Information Hiding : Example

- Regardless of how `Display()` extracts the current timestamp from `ticks`, users can be sure that it will "do the right thing" and display the correct time onscreen.

Functional independence

- Achieved by developing independent modules – each module address a specific subset of requirements.
- Is assessed using cohesion and coupling.
- Cohesion - is an indication of the relative functional strength of a module.
  - a cohesive module should (ideally) do just one thing.
- Coupling - is an indication of the relative interdependence among modules.
  - depends on the interface complexity between modules, the point at which entry or reference is made to a module, and what data pass across the interface.

Composite/Structured Design

- Method for breaking up a product into modules for
  - Maximal interaction within module, and
  - Minimal interaction between modules
- Module cohesion
  - Degree of interaction within a module
- Module coupling
  - Degree of interaction between modules

Cohesion

- Degree of interaction within a module
- Seven categories or levels of cohesion (non-linear scale)

7.	Informational cohesion	(Good)
	Functional cohesion	
5.	Communicational cohesion	
4.	Procedural cohesion	
3.	Temporal cohesion	
2.	Logical cohesion	
1.	Coincidental cohesion	(Bad)

Coincidental Cohesion

- A module has coincidental cohesion if it performs multiple, completely unrelated actions
- Example
  - print next line, reverse string of characters comprising second parameter, add 7 to fifth parameter, convert fourth parameter to floating point
- Arise from rules like
  - “Every module will consist of between 35 and 50 statements”

Why Coincidental Cohesion is BAD?

- Degrades maintainability
- Modules are not reusable
- This is easy to fix
  - Break into separate modules each performing one task

Logical Cohesion

- A module has logical cohesion when it performs a series of related actions, one of which is selected by the calling module
- Example 1

```
function code = 7;
new operation (op code, dummy 1, dummy 2, dummy 3);
// dummy 1, dummy 2, and dummy 3 are dummy variables,
// not used if function code is equal to 7
```

- Example 2
  - Module performing all input and output
- Example 3
  - One version of OS/VS2 contained logical cohesion module performing 13 different actions. Interface contained 21 pieces of data

Why Logical Cohesion is BAD?

- The interface is difficult to understand
- Code for more than one action may be intertwined
- Difficult to reuse

Why Logical Cohesion is BAD?

- A new tape unit is installed
- What is the effect on the laser printer?

1. Code for all input and output
2. Code for input only
3. Code for output only
4. Code for disk and tape I/O
5. Code for disk I/O
6. Code for tape I/O
7. Code for disk input
8. Code for disk output
9. Code for tape input
10. Code for tape output
⋮ ⋮ ⋮
37. Code for keyboard input

Temporal Cohesion

- A module has temporal cohesion when it performs a series of actions related in time
- Example
  - open old master file, new master file, transaction file, print file, initialize sales district table, read first transaction record, read first old master record (a.k.a. perform initialization)

Why Temporal Cohesion is BAD?

- Actions of this module are weakly related to one another, but strongly related to actions in other modules.
  - Consider sales district table
- Not reusable

Procedural Cohesion

- A module has procedural cohesion if it performs a series of actions related by the procedure to be followed by the product
- Example
  - read part number and update repair record on master file

Why Procedural Cohesion is BAD?

- Actions are still weakly connected, so module is not reusable

Communicational Cohesion

- A module has communicational cohesion if it performs a series of actions related by the procedure to be followed by the product, but in addition all the actions operate on the same data
- Example 1
  - update record in database and write *it* to audit trail
- Example 2
  - calculate new coordinates and send *them* to terminal

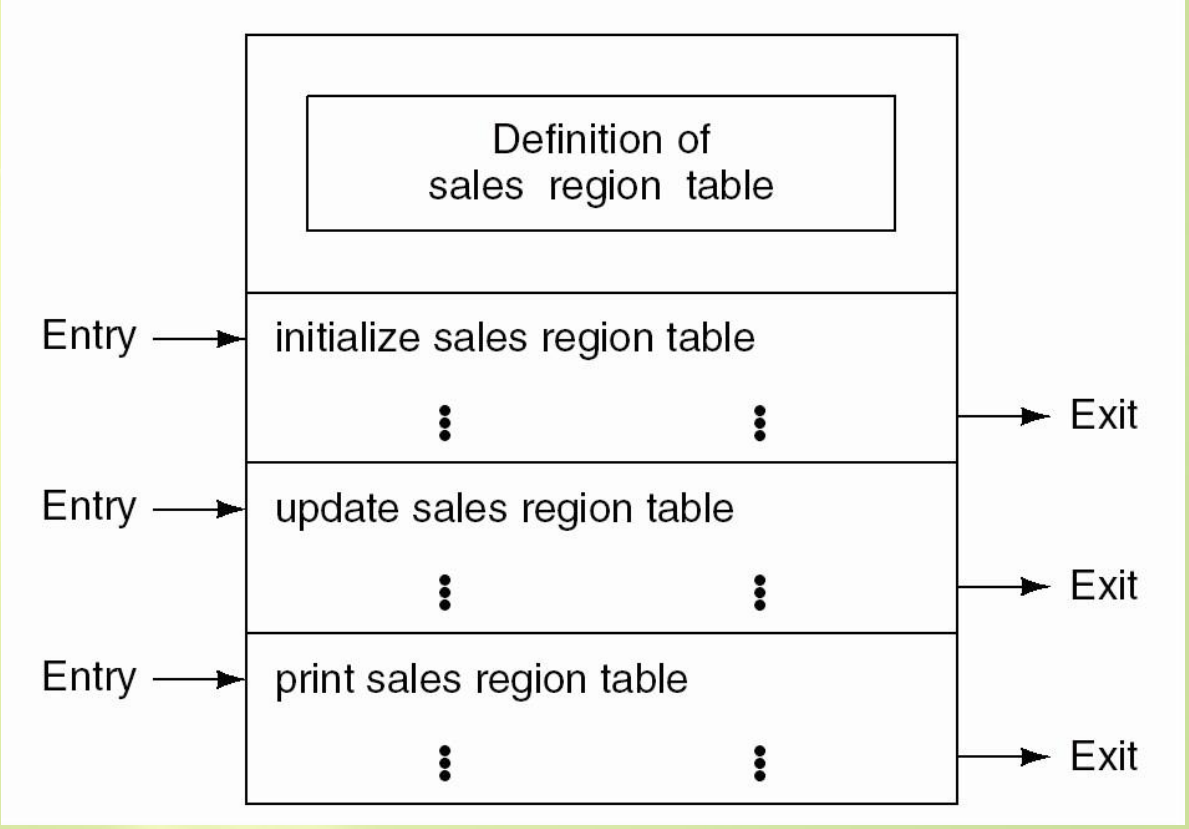
Why is Communicational Cohesion so BAD?

- Still lack of reusability

Informational Cohesion

- A module has informational cohesion if it performs a number of actions, each with its own entry point, with independent code for each action, all performed on the same data structure

Why is Informational Cohesion is GOOD?



- Essentially, this is an abstract data type

Functional Cohesion

- Module with functional cohesion performs exactly one action
- Example 1
  - get temperature of furnace
- Example 2
  - compute orbital of electron
- Example 3
  - write to flash drive
- Example 4
  - calculate sales commission

Why is Functional Cohesion is GOOD?

- More reusable
- Corrective maintenance easier
  - Fault isolation
  - Fewer regression faults
- Easier to extend product

Coupling

- Degree of interaction between two modules
- Five categories or levels of coupling (non-linear scale)

5.	Data coupling	(Good)
4.	Stamp coupling	
3.	Control coupling	
2.	Common coupling	
1.	Content coupling	(Bad)

Content Coupling

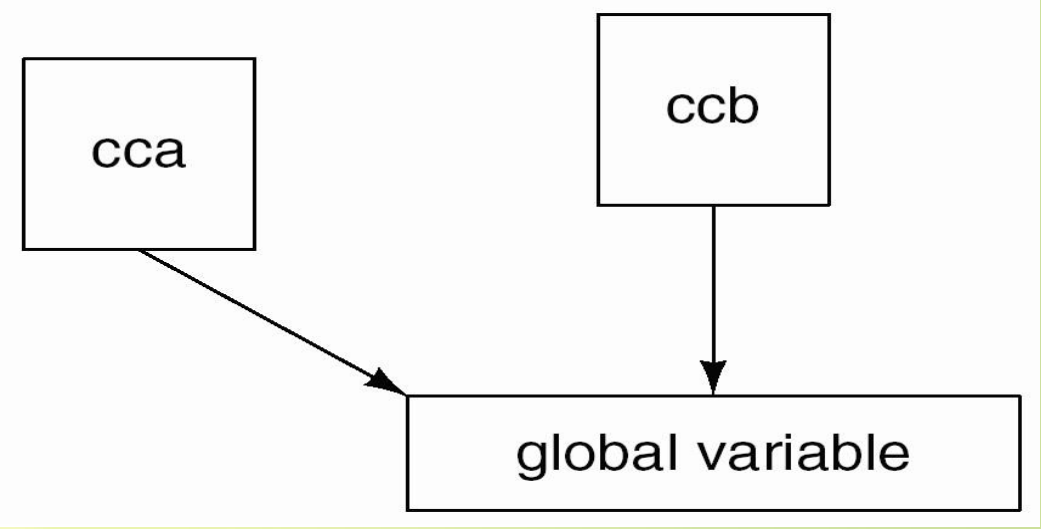
- Two modules are content coupled if one directly references contents of the other
- Example 1
  - Module a modifies statement of module b
- Example 2
  - Module a refers to local data of module b in terms of some numerical displacement within b
- Example 3
  - Module a branches into local label of module b

Why is Content Coupling so BAD?

- Almost any change to b, even recompiling b with new compiler or assembler, requires change to a
- Warning
  - Content coupling can be implemented in Ada through use of overlays implemented via address clauses

Common Coupling

- Two modules are common coupled if they have write access to global data



- Example 1
  - Modules cca and ccb can access ***and change*** value of global variable

Common Coupling

- Example 2
  - Modules cca and ccb both have access to same database, and can both read *and write* same record

Why Common Coupling is so BAD?

- Contradicts the spirit of structured programming
  - The resulting code is virtually unreadable

```
while (global variable == 0)
{
   if (argument xyz > 25)
      module 3 ();
   else
      module 4 ();
}
```

Why Common Coupling is so BAD?

- Modules can have side-effects
  - This affects their readability
- Entire module must be read to find out what it does
- Difficult to reuse
- Module exposed to more data than necessary

Control Coupling

- Two modules are control coupled if one passes an element of control to the other
- Example 1
  - Operation code passed to module with logical cohesion
- Example 2
  - Control-switch passed as argument

Why Control Coupling is so BAD?

- Modules are not independent; module b (the called module) must know internal structure and logic of module a.
  - Affects reusability
- Associated with modules of logical cohesion

Stamp Coupling

- Some languages allow only simple variables as parameters
  - part number
  - satellite altitude
  - degree of multiprogramming
- Many languages also support passing of data structures
  - part record
  - satellite coordinates
  - segment table

Stamp Coupling

- Two modules are stamp coupled if a data structure is passed as a parameter, but the called module operates on some but not all of the individual components of the data structure

Why is Stamp Coupling so BAD?

- It is not clear, without reading the entire module, which fields of a record are accessed or changed
  - Example

    calculate withholding (employee record)
- Difficult to understand
- Unlikely to be reusable
- More data than necessary is passed
  - Uncontrolled data access can lead to computer crime
- There is nothing wrong with passing a data structure as a parameter, provided *all* the components of the data structure are accessed and/or changed

  invert matrix (original matrix, inverted matrix);

  print inventory record (warehouse record);

Data Coupling

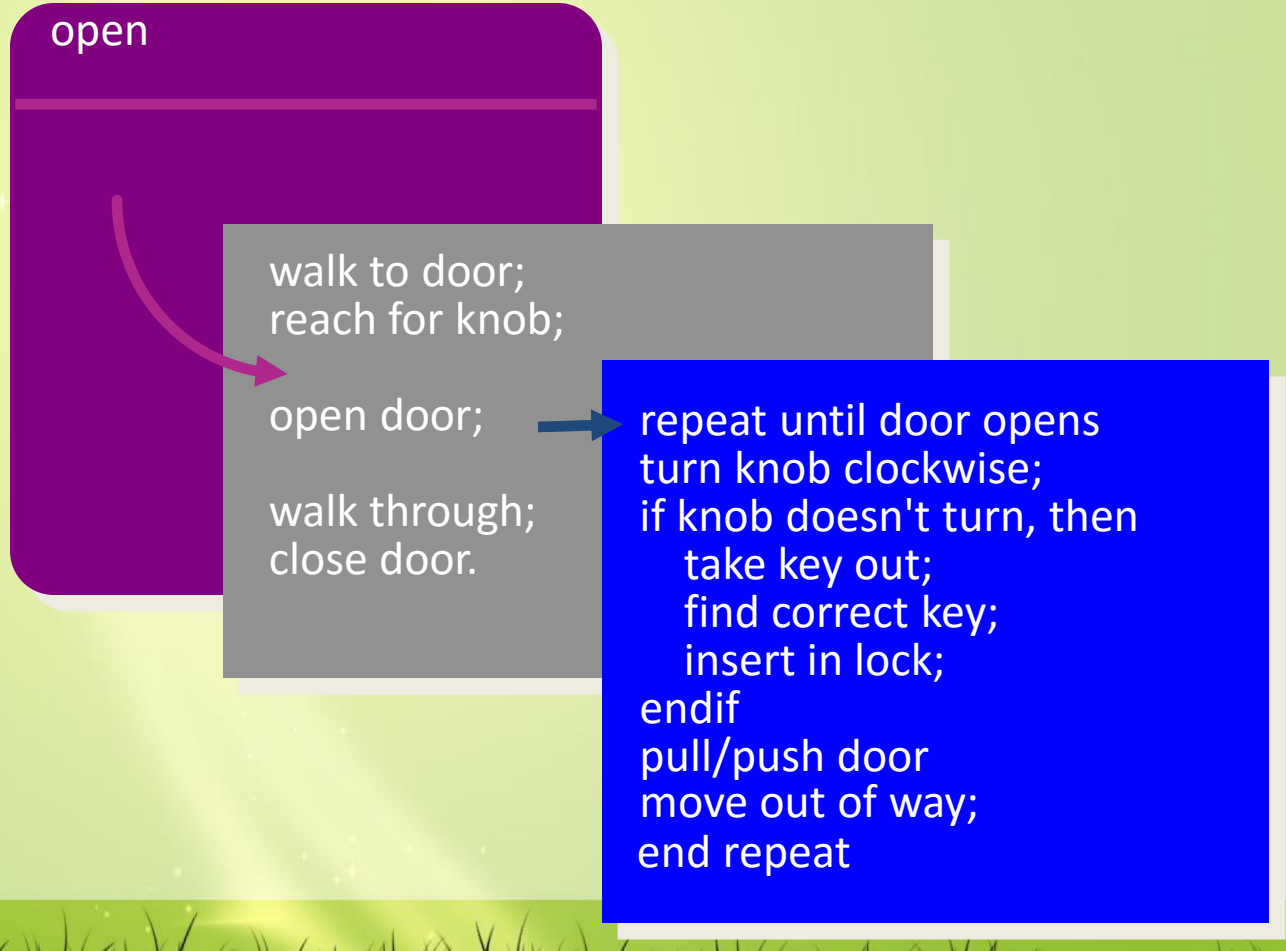
- Two modules are data coupled if all parameters are homogeneous data items [simple parameters, or data structures all of whose elements are used by called module]

- Examples
  - display time of arrival (flight number);
  - compute product (first number, second number);
  - get job with highest priority (job queue);

Why is Data Coupling so GOOD?

- The difficulties of content, common, control, and stamp coupling are not present
- Maintenance is easier

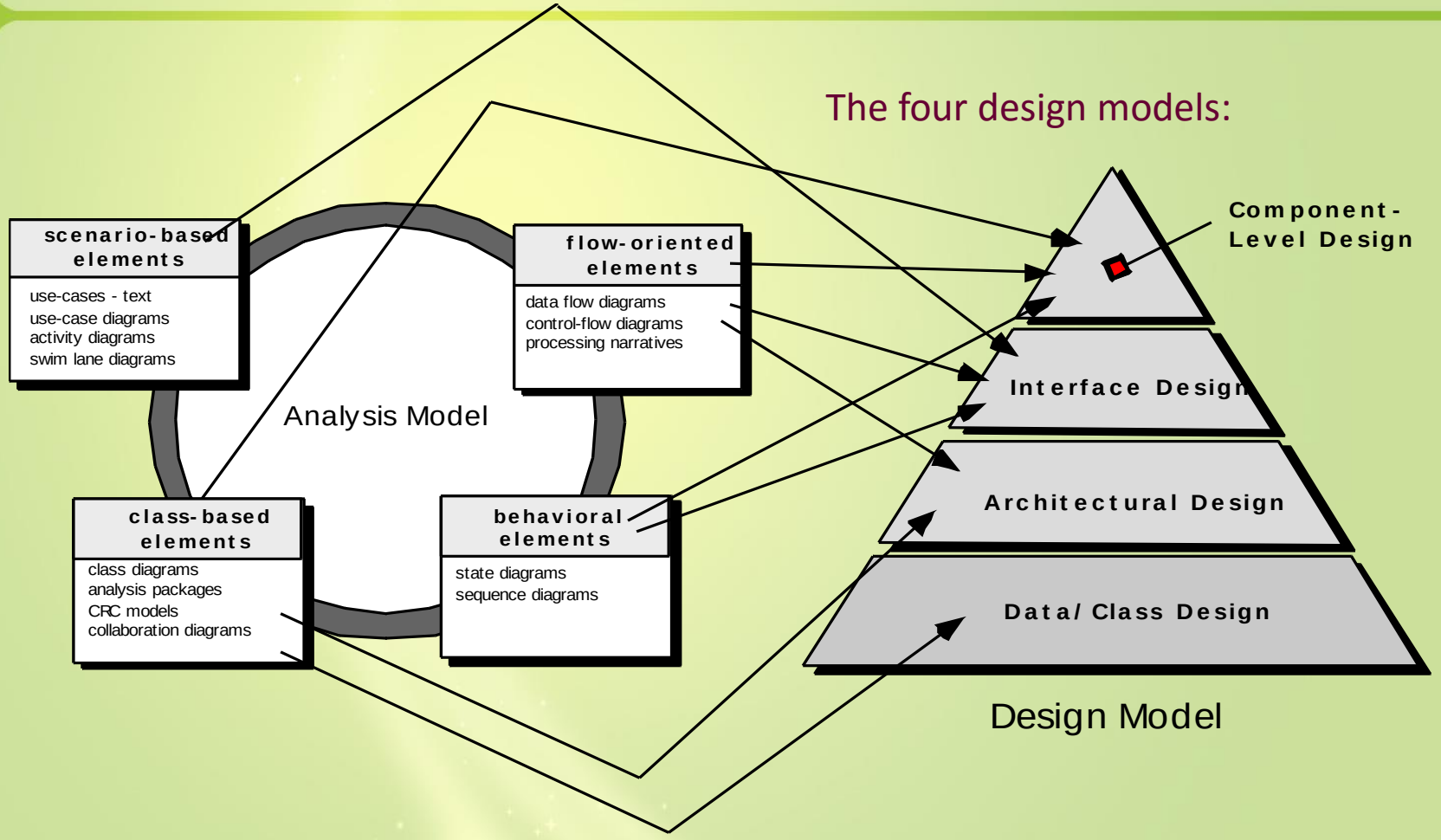
Refinement


open
walk to door;
reach for knob;
open door;
walk through;
close door.
repeat until door opens
turn knob clockwise;
if knob doesn't turn, then
take key out;
find correct key;
insert in lock;
endif
pull/push door
move out of way;
end repeat

Refactoring

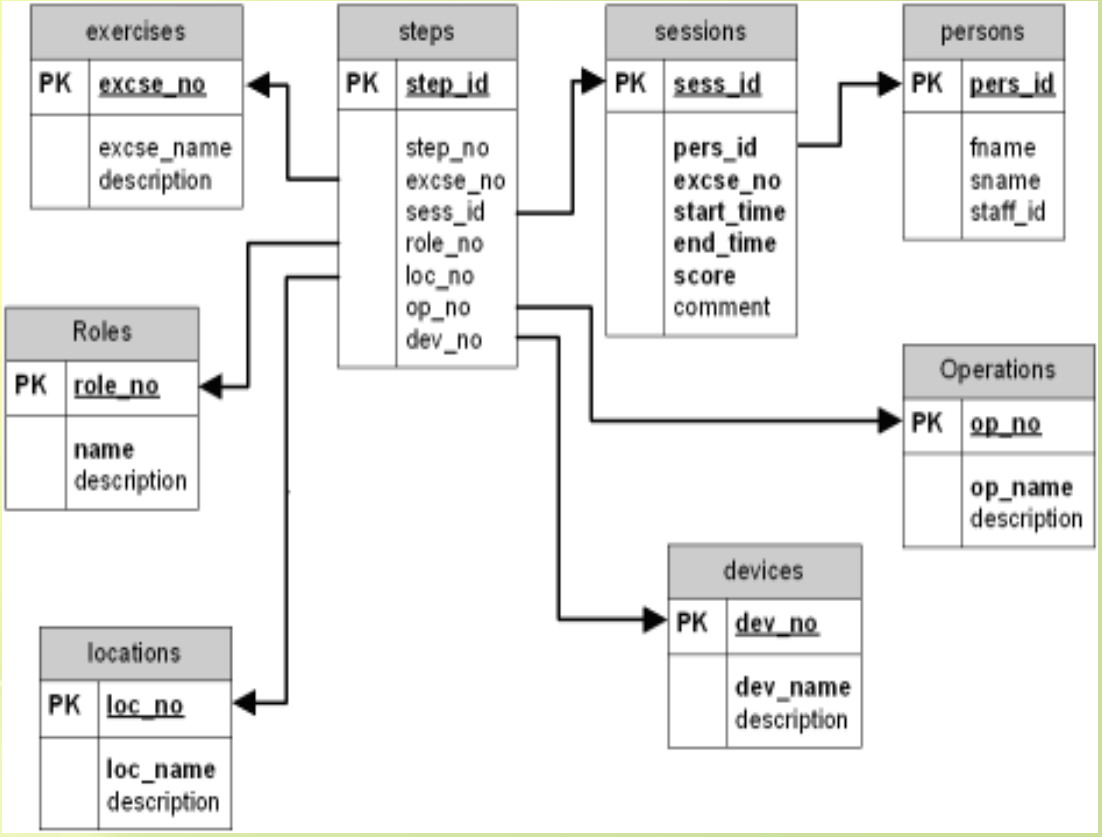
- According to Fowler (1999):
  - "Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code [design] yet improves its internal structure."
- When software is refactored, the existing design is examined for
  - redundancy
  - unused design elements
  - inefficient or unnecessary algorithms
  - poorly constructed or inappropriate data structures
  - or any other design failure that can be corrected to yield a better design.

Analysis Models → Design Model



Data/Class Design Model

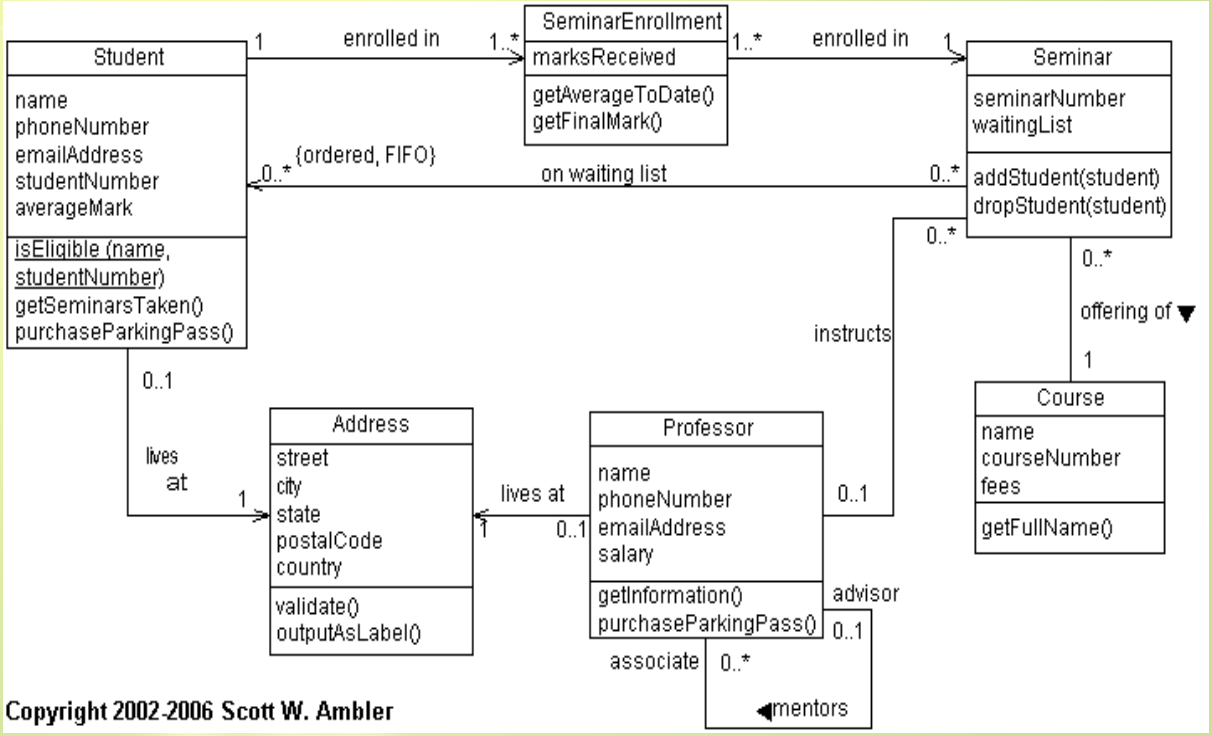
example: ERD



Source: http://www.svgopen.org/2003/papers/SvgInterfaceElectricalSwitching/index.html

Data/Class Design Model

example: Class Diagram



Source: http://www.agiledata.org/essays/objectOrientation101.html

Architectural Design Elements

The architectural model [Sha96] is derived from three sources:

- information about the application domain for the software to be built;
- specific requirements model elements such as data flow diagrams or analysis classes, their relationships and collaborations for the problem at hand, and
- the availability of architectural patterns and styles.

Why Architecture?

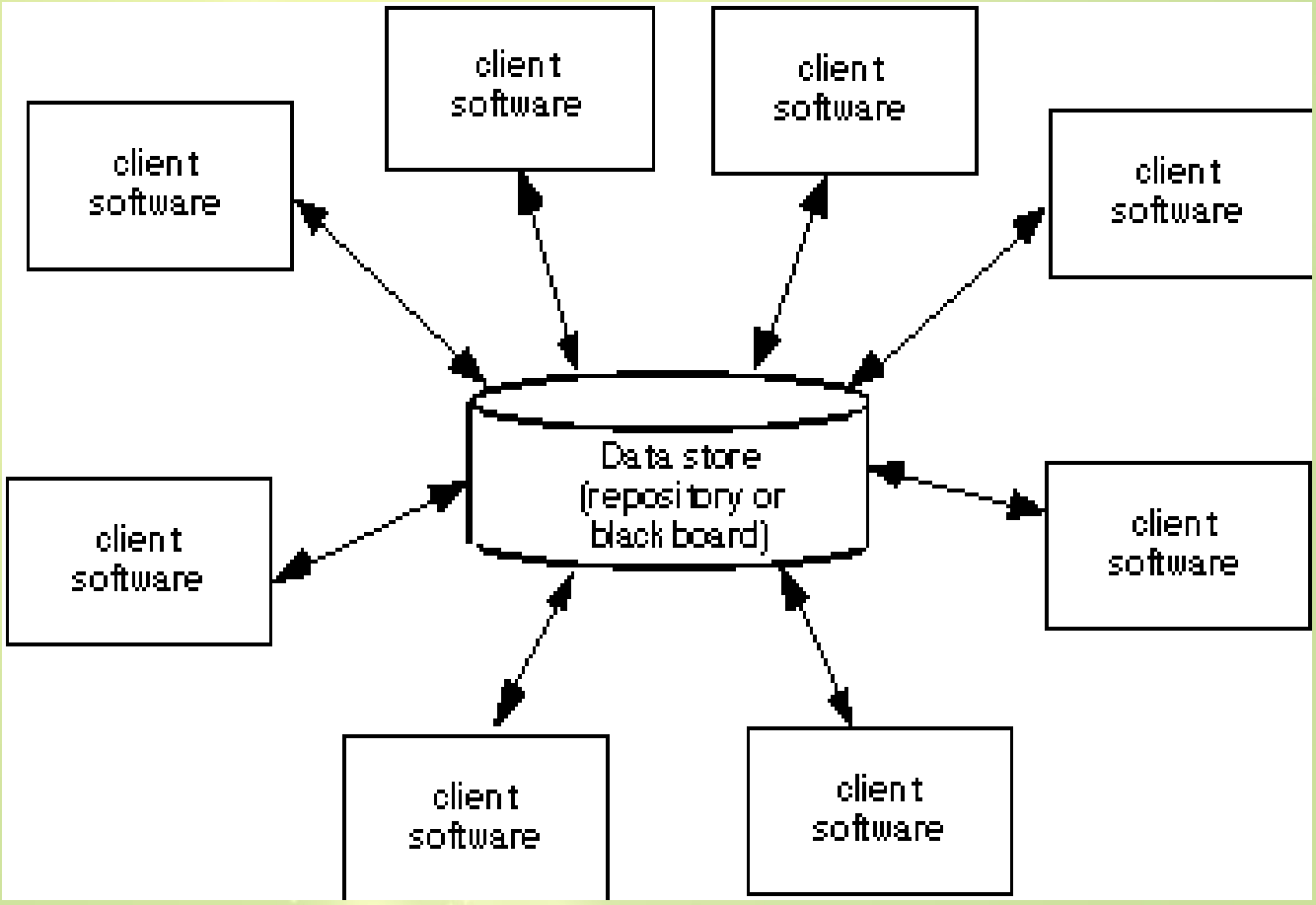
The architecture is not the operational software. Rather, it is a representation that enables a software engineer to:

1. analyze the effectiveness of the design in meeting its stated requirements,
2. consider architectural alternatives at a stage when making design changes is still relatively easy, and
3. reduce the risks associated with the construction of the software.

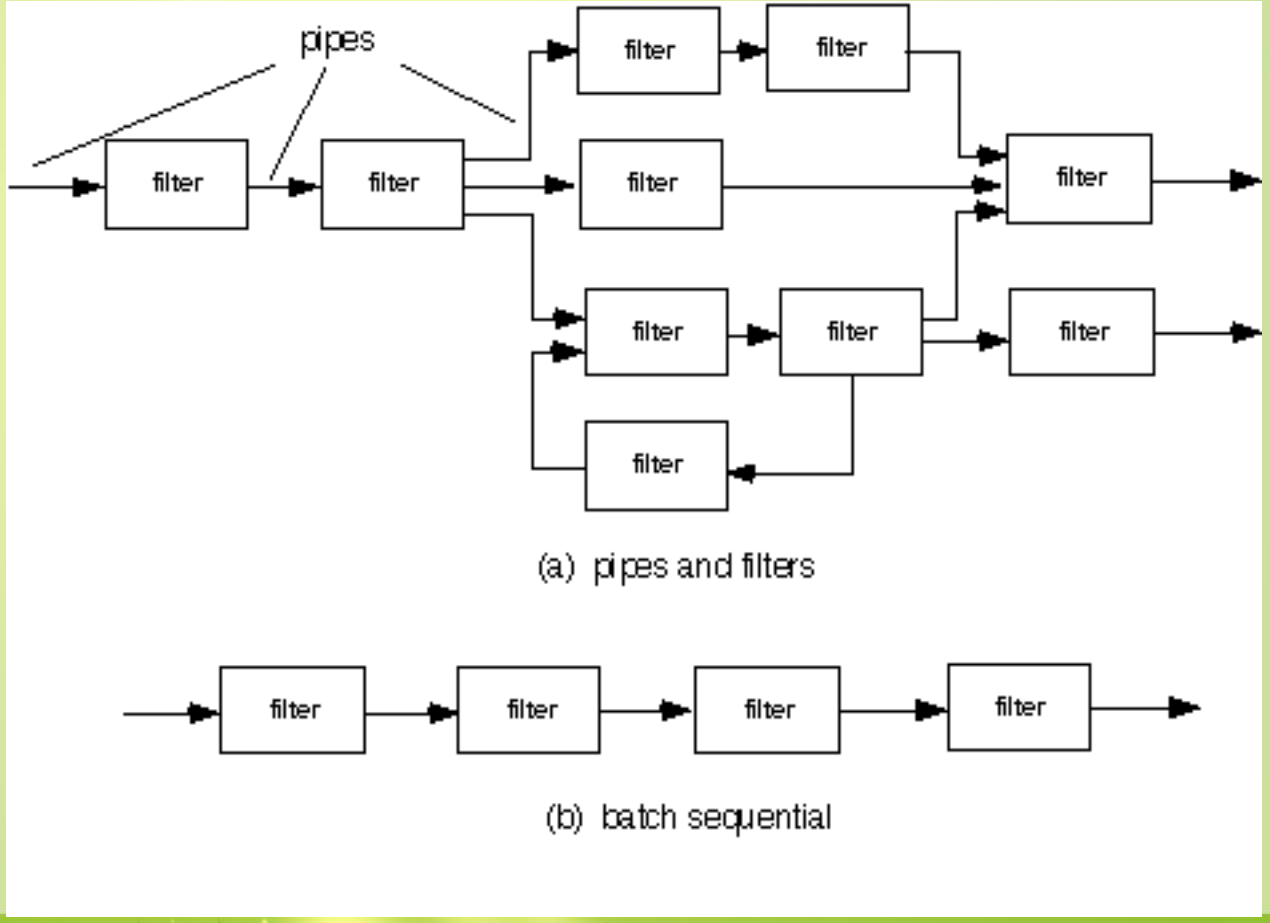
Architecture Styles

- Data-centered architectures
- Data flow architectures
- Call and return architectures
- Object-oriented architectures
- Layered architectures

Data Centered Architecture



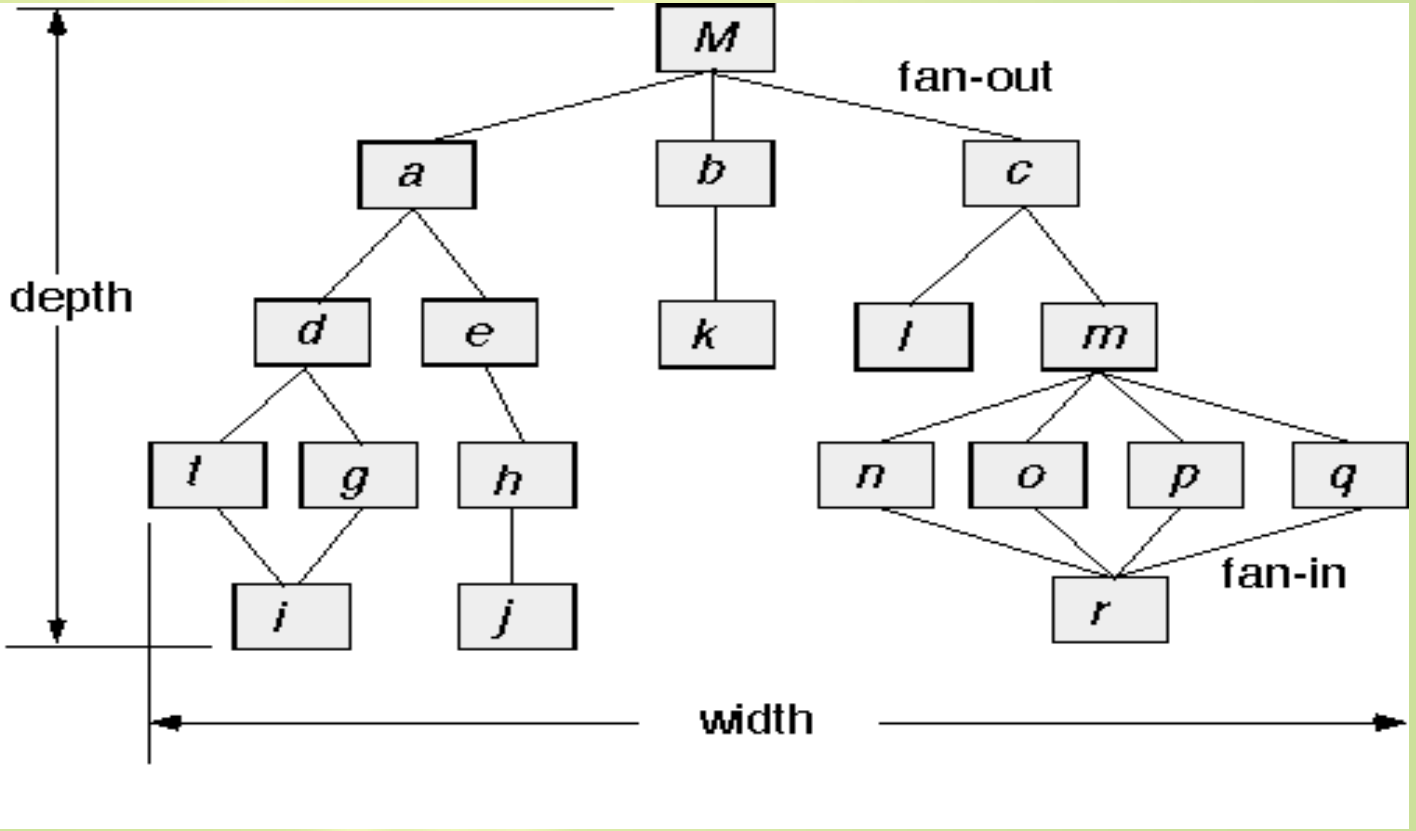
Data Flow Architecture



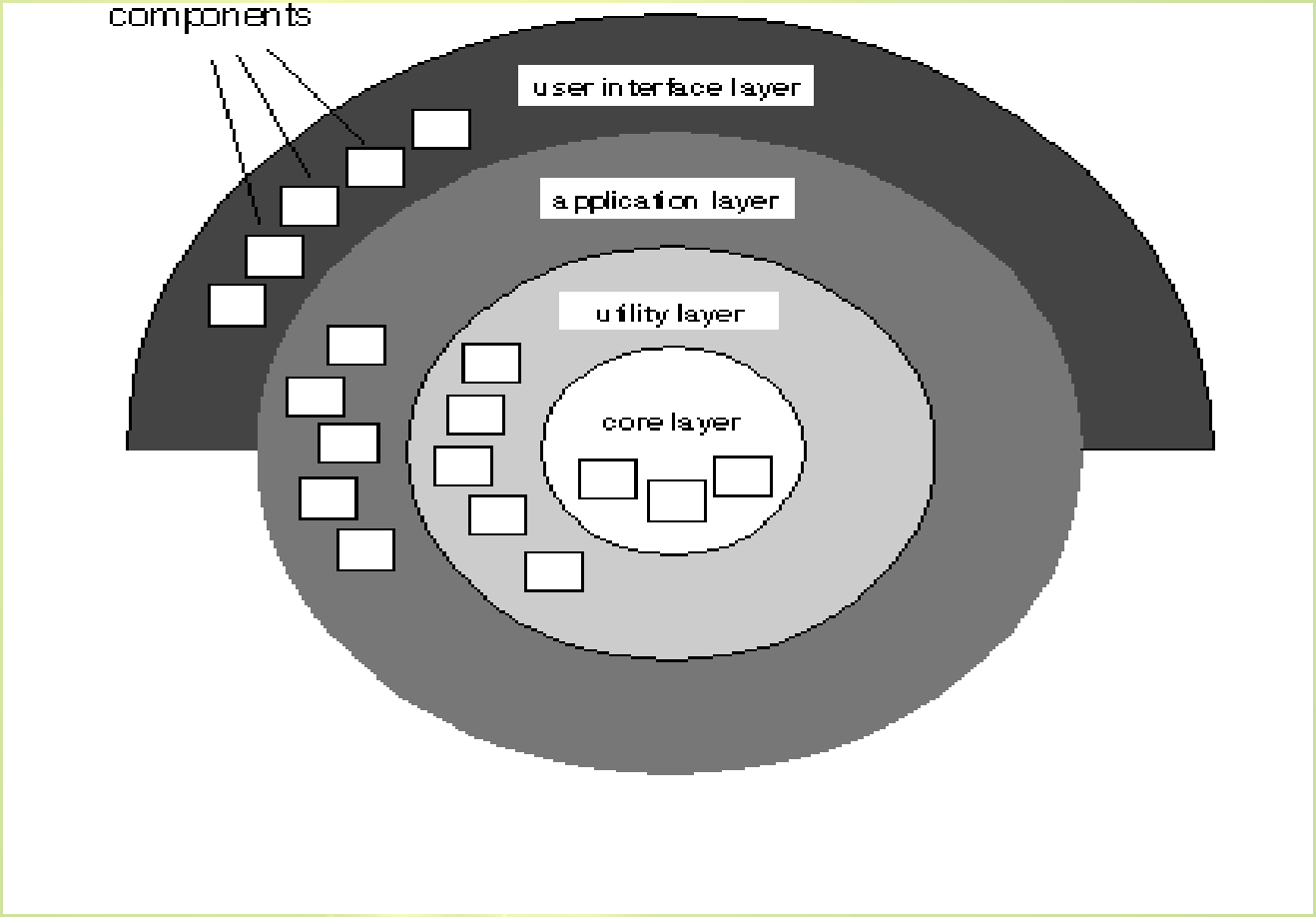
(a) pipes and filters

(b) batch sequential

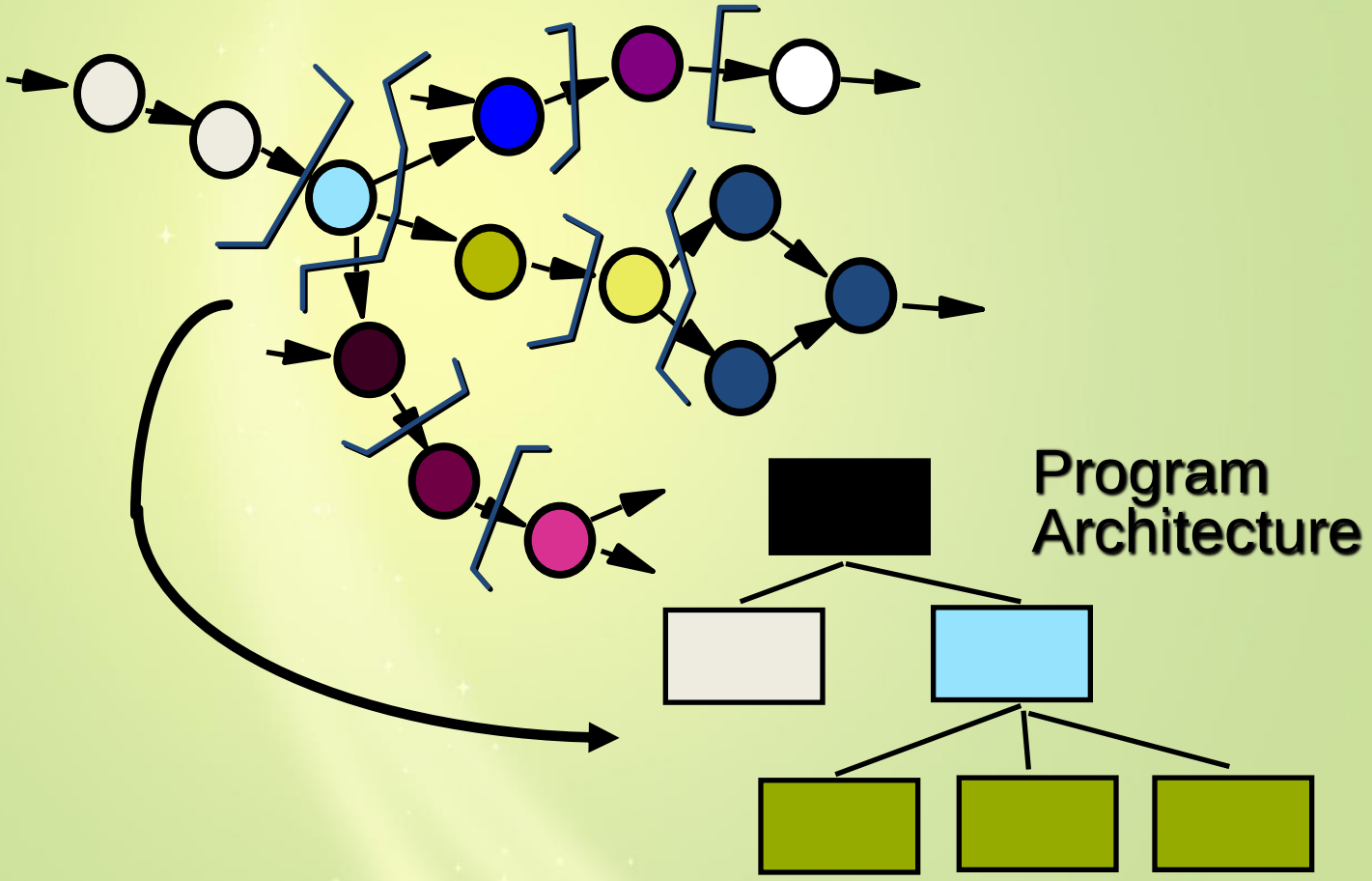
Call and Return Architecture?



Layered Architecture

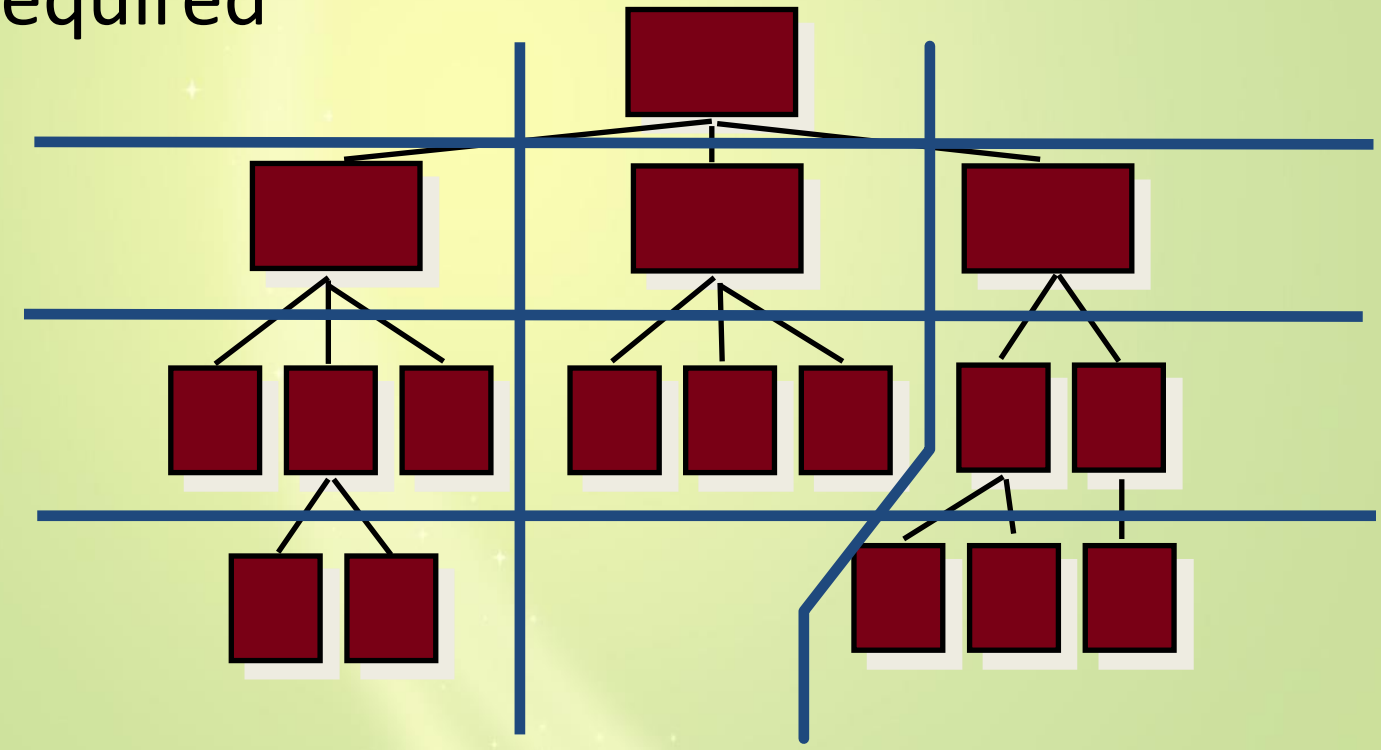


Deriving Program Architecture



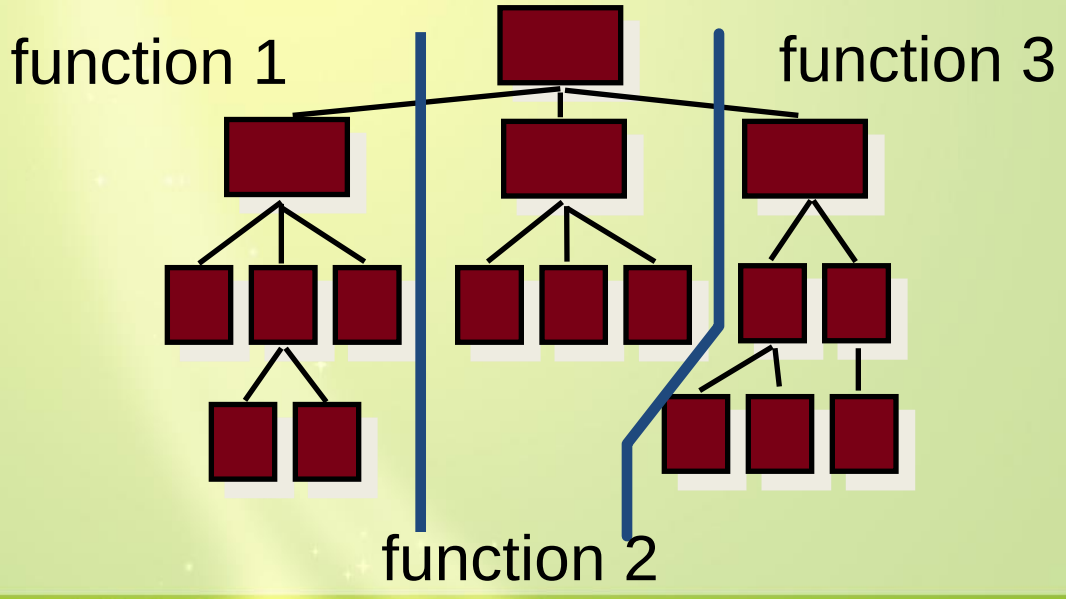
Partitioning the Architecture

- “horizontal” and “vertical” partitioning are required



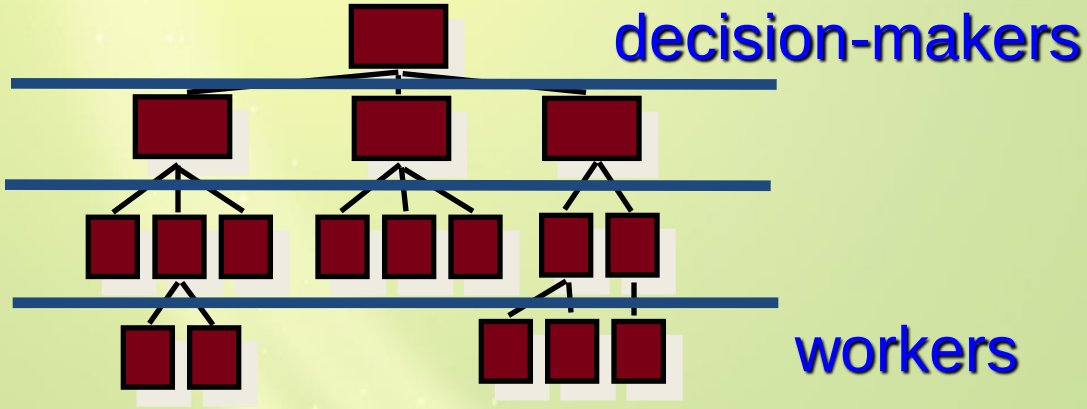
Horizontal Partitioning

- define separate branches of the module hierarchy for each major function
- use control modules to coordinate communication between functions



Vertical Partitioning

- design so that decision making and work are stratified
- decision making modules should reside at the top of the architecture



Why Partitioned Architecture?

- results in software that is easier to test
- leads to software that is easier to maintain
- results in propagation of fewer side effects
- results in software that is easier to extend

Structured Design

- **objective**: to derive a program architecture that is partitioned
- **approach:**
  - the DFD is mapped into a program architecture
  - the PSPEC and STD are used to indicate the content of each module
- **notation**: structure chart

User Interface Design

Interface Design

Easy to learn?

Easy to use?

Easy to understand?

Interface Design

Typical Design Errors

lack of consistency
too much memorization
no guidance / help
no context sensitivity
poor response
Arcane/unfriendly

Golden Rules

- Place the user in control
- Reduce the user's memory load
- Make the interface consistent

Place the User in Control

- Define interaction modes in a way that does not force a user into unnecessary or undesired actions.
- Provide for flexible interaction.
- Allow user interaction to be interruptible and undoable.
- Streamline interaction as skill levels advance and allow the interaction to be customized.
- Hide technical internals from the casual user.
- Design for direct interaction with objects that appear on the screen.

Reduce the User's Memory

- Reduce demand on short-term memory.
- Establish meaningful defaults.
- Define shortcuts that are intuitive.
- The visual layout of the interface should be based on a real world metaphor.
- Disclose information in a progressive fashion.

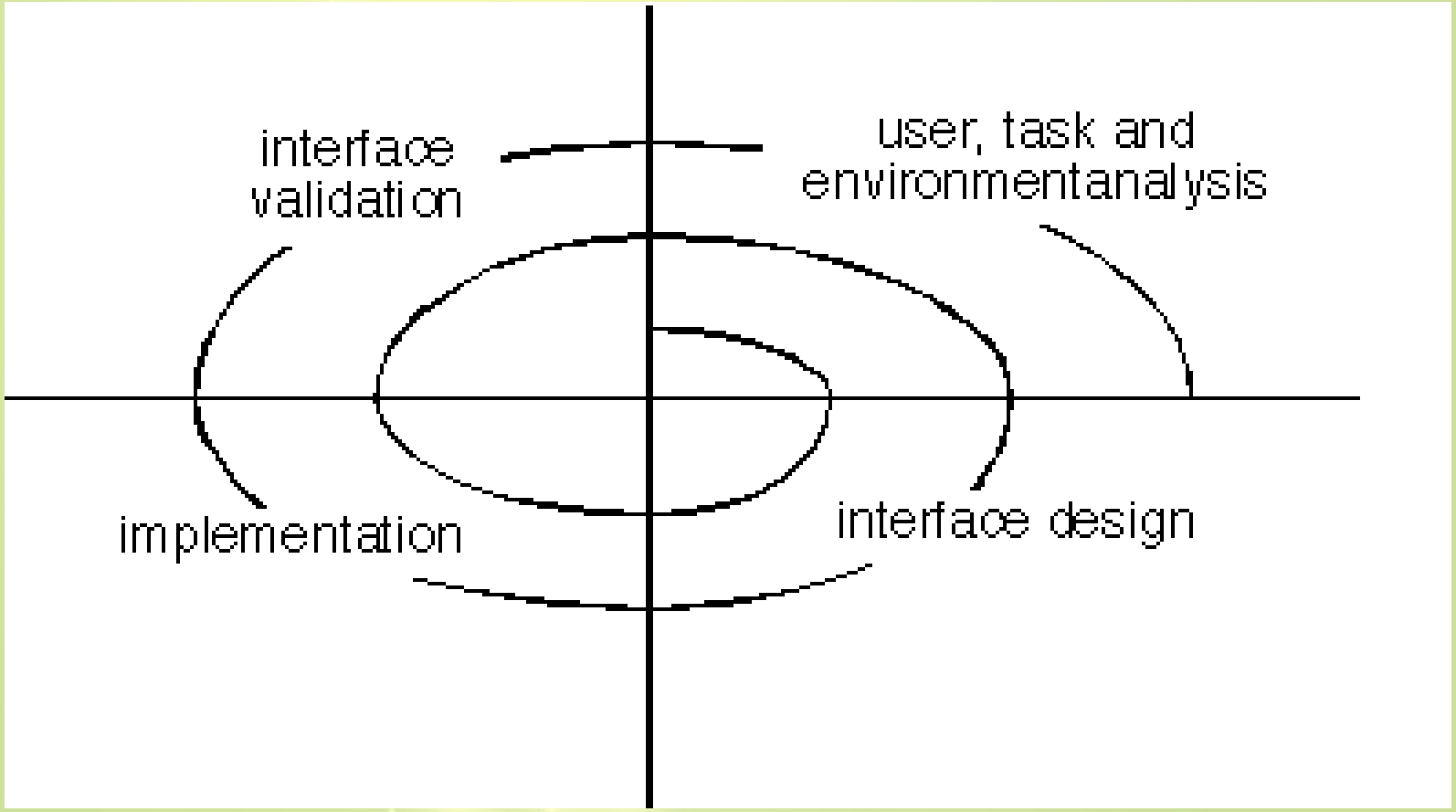
Make the User Interface Consistent

- Allow the user to put the current task into a meaningful context.
- Maintain consistency across a family of applications.
- If past interactive models have created user expectations, do not make changes unless there is a compelling reason to do so.

User Interface Design Models

- System perception — the user's mental image of what the interface is
- User model — a profile of all end users of the system
- System image — the "presentation" of the system projected by the complete interface
- Design model — data, architectural, interface and procedural representations of the software

User Interface Design Process



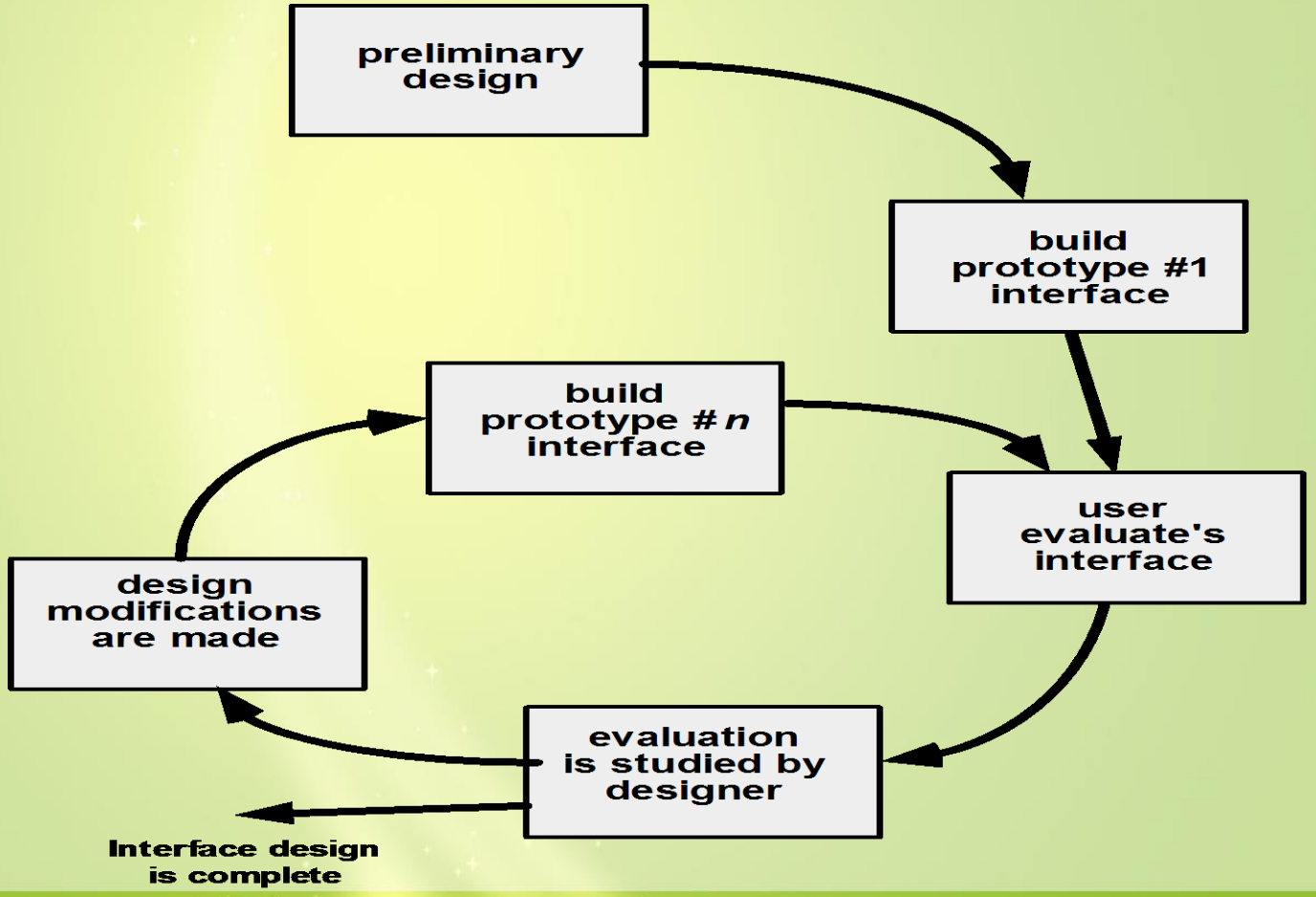
Task Analysis and Modeling

- All human tasks required to do the job (of the interface) are defined and classified
- Objects (to be manipulated) and actions (functions applied to objects) are identified for each task
- Tasks are refined iteratively until the job is completely defined

Interface Design Activities

1. Establish the goals and intentions for each task.
2. Map each goal/intention to a sequence of specific actions.
3. Specify the action sequence of tasks and subtasks, also called a user scenario, as it will be executed at the interface level.
4. Indicate the state of the system, i.e., what does the interface look like at the time that a user scenario is performed?
5. Define control mechanisms, i.e., the objects and actions available to the user to alter the system state.
6. Show how control mechanisms affect the state of the system.
7. Indicate how the user interprets the state of the system from information provided through the interface.

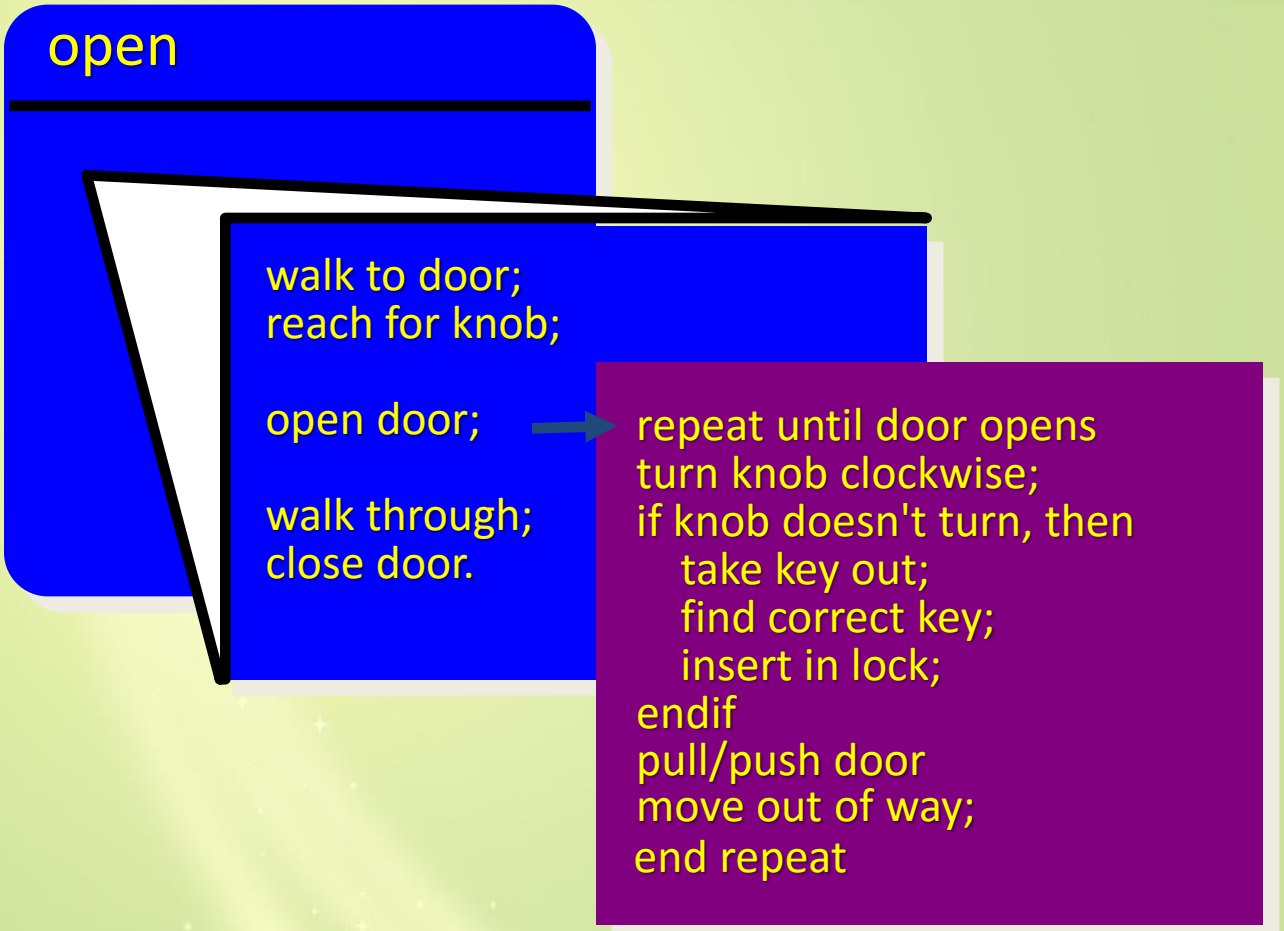
Design Evaluation Cycle



Component Level Design

- the closest design activity to coding
- the approach:
  - review the design description for the component
  - use stepwise refinement to develop algorithm
  - use structured programming to implement procedural logic
  - use ‘formal methods’ to prove logic

Stepwise Refinement



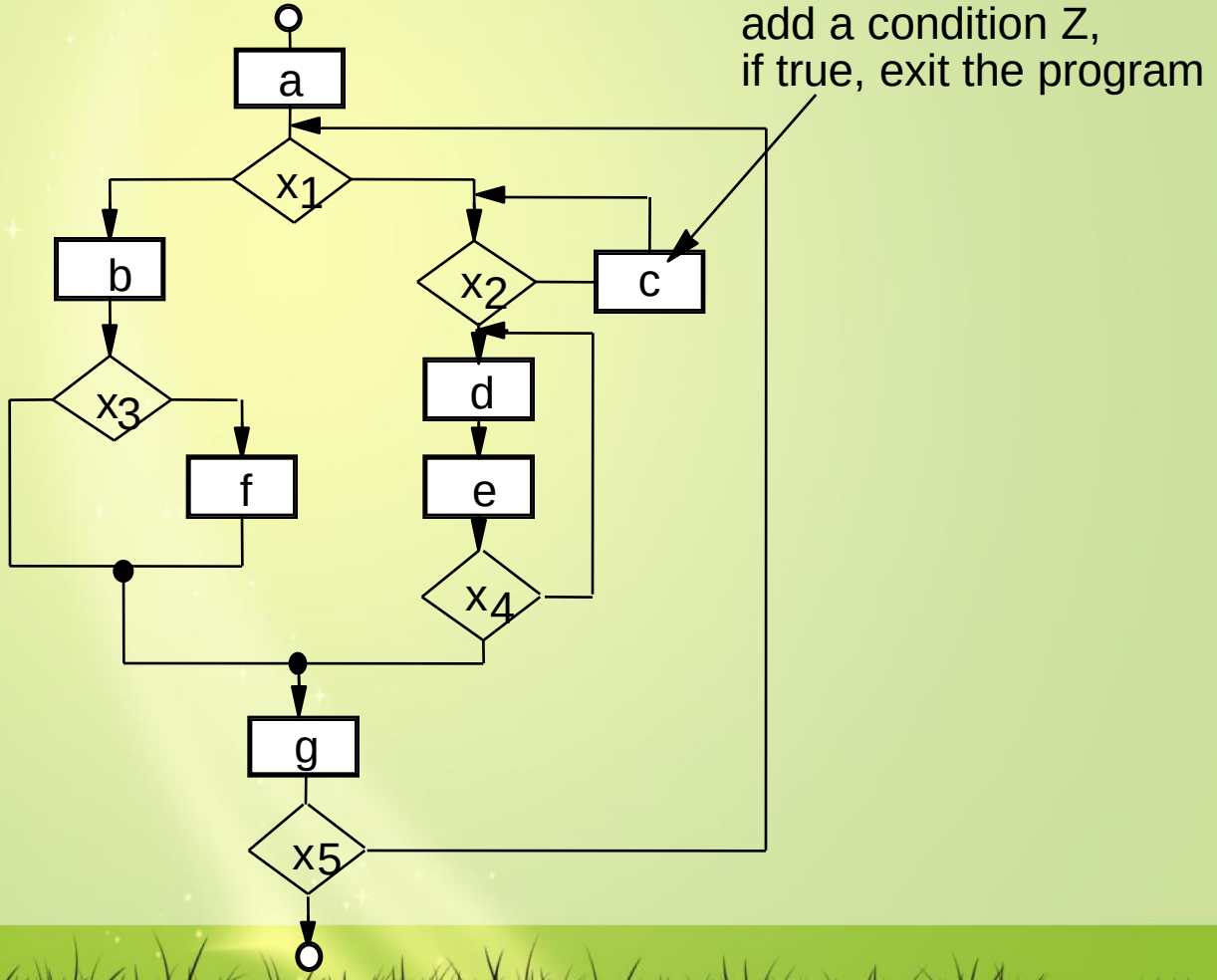
Component Level Design Model

- represents the algorithm at a level of detail that can be reviewed for quality
- options:
  - graphical (e.g. flowchart, box diagram)
  - pseudocode (e.g., PDL) … choice of many
  - programming language
  - decision table
  - conduct walkthrough to assess quality

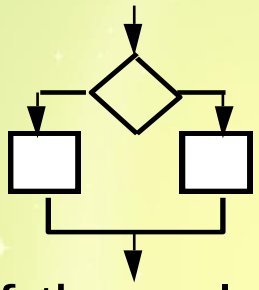
Structured Programming for Procedural Design

- uses a limited set of logical constructs:
  - *sequence*
  - *conditional* — if-then-else, select-case
  - *loops* — do-while, repeat until
- leads to more readable, testable code
- important for achieving high quality, but not enough

A Structured Procedural Design



Program Design Language (PDL)



if-then-else

```
if condition x
  then process a;
  else process b;
endif
```

PDL

- easy to combine with source code
- machine readable, no need for graphics input
- graphics can be generated from PDL
- enables declaration of data as well as procedure
- easier to maintain

Why Design Language?

- can be a derivative of the HOL of choice e.g., Ada PDL
- machine readable and processable
- can be embedded with source code, therefore easier to maintain
- can be represented in great detail, if designer and coder are different
- easy to review

Design Evaluation

- "A good software design minimizes the time required to create, modify, and maintain the software while achieving run-time performance.“ (Shore and Chromatic, 2007)
- According to Elssamadisy (2007):

1) Design quality is people-sensitive.
   - ✓ For instance, design quality is dependent upon the programmers writing and maintaining the code.

2) Design quality is change-specific.
   - ✓ There are two general ways to make designs of higher quality with respect to this aspect:
     1. generalizing from the tools like design patterns, and
     2. using tests and refactoring as change-enablers.

Design Evaluation

3) Modification and maintenance time are more important than creation time.

- ✓ Because:
  1. time spent in maintenance is much more than creation time of a software, and
  2. in iterative development, modification happens during the initial creation of the software.

4) Design quality is unpredictable.

- ✓ Because quality is really dependant on the team developing the software. As the team changes, or evolves, then the design quality also evolves.
- ✓ You really only know how good a design is by it standing the test of time and modifications.

Design Evaluation

- **Points to ponder:**
  - ✓ to maintain the quality of the design, we must maintain the theory of the design as the programming team evolves,
    - That design quality is tied to the people who are building and maintaining the software(Shore and Chromatic, 2007).

Design Specification

- IEEE Standard for Information Technology -Systems Design - Software Design Descriptions (IEEE 1016-2009) an improved version of the 1998 version.
- This standard specifies an organizational structure for a software design description (SDD).
  - An SDD is a document used to specify system architecture and application design in a software related project.
  - Provides a generic template for an SDD.