UNIT -3

Software Design

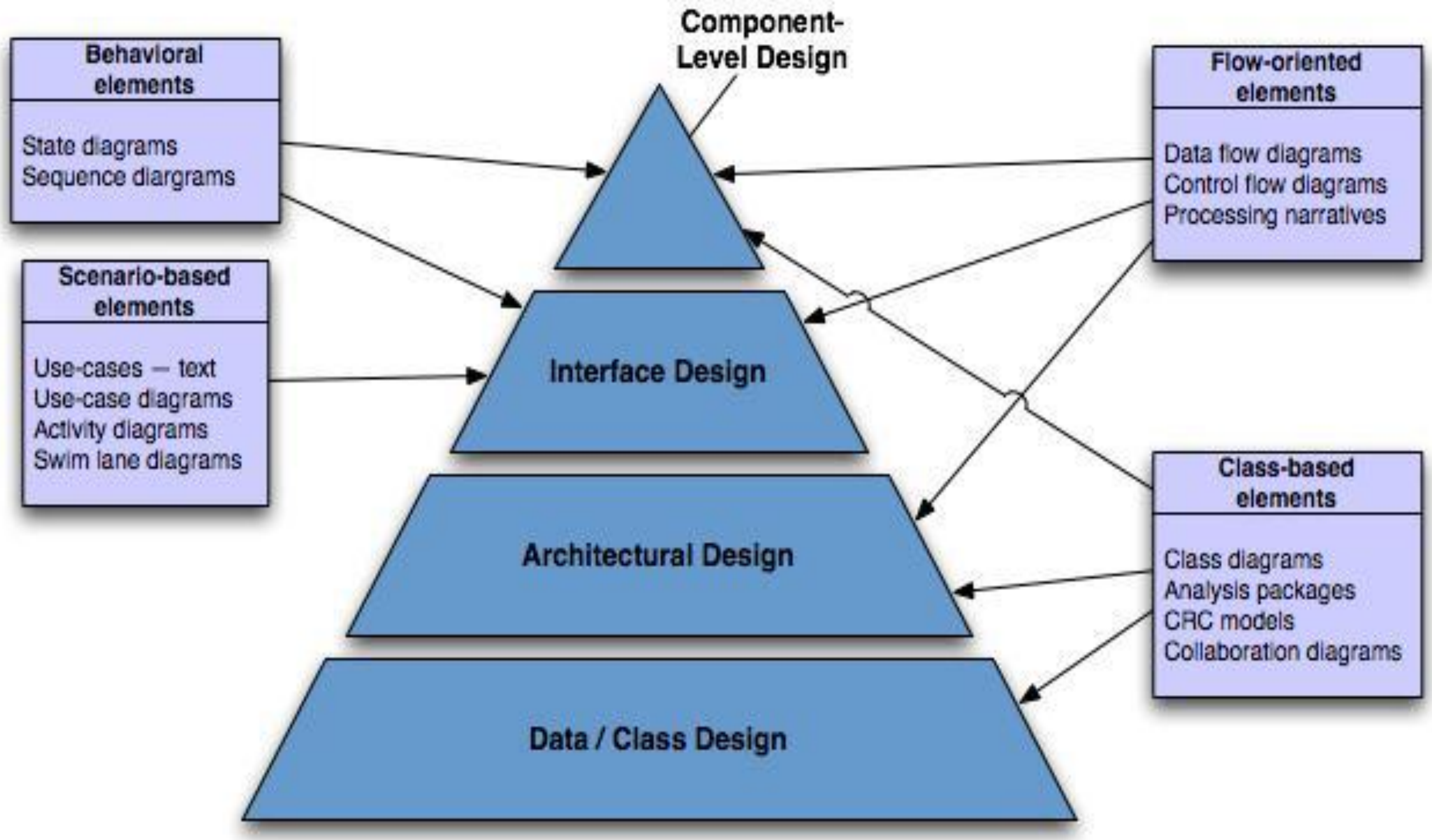
Design Engineering

- It covers the set of principles, concepts, and practices that lead to the development of a high quality system or product.
- Goal of design engineering is to produce a model or representation that depict:
  - Firmness – program should not have any bug that inhibits its functions.
  - Commodity – suitable to its intended use.
  - Delight - pleasurable to use
- The design model provides detail about software data structures, architecture, interfaces, and components that are necessary to implement the system.

Software Design

- Software design model consists of 4 designs:
  - Data/class Design
  - Architectural Design
  - Interface Design
  - Component Design

Translating Analysis → Design



- **Data/class design** - Created by transforming the analysis model class-based elements into classes and data structures required to implement the software
- **Architectural design** - defines the relationships among the major structural elements of the software, it is derived from the class-based elements and flow-oriented elements of the analysis model
- **Interface design** - describes how the software elements, hardware elements, and end-users communicate with one another, it is derived from the analysis model scenario-based elements, flow-oriented elements, and behavioral elements
- **Component-level design** - created by transforming the structural elements defined by the software architecture into a procedural description of the software components using information obtained from the analysis model class-based elements, flow-oriented elements, and behavioral elements

Why design is so important?

- It is place where quality is fostered.
- It provides us with representation of software that can be assessed for quality.
- Only way that can accurately translate a customer’s requirements into a finished software product.
- It serves as foundation for all software engineering activities.
- Without design difficult to assess:
  - Risk
  - Test
  - Quality

Design Process and Design Quality

- S/w design is an iterative process through which requirements are translated into a “blueprint” for constructing the s/w.
- As design iteration occur, subsequent refinement leads to design representation at much lower levels of abstraction.

Goal of design process

- The design 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.
- The design must be a readable, understandable guide for those who generate code and for those who test and subsequently support the software.
- The design should provide a complete picture of the software, addressing the data, functional, and behavioral domains from an implementation perspective.

Quality Guidelines

Characteristics of good design

- A design should exhibit an architecture that
    - as been created using recognizable architectural styles or patterns,
    - is composed of components that exhibit good design characteristics and
    - can be implemented in an evolutionary fashion
- A design should be modular; that is, the software should be logically partitioned into elements or subsystems
- A design should contain distinct representations of data, architecture, interfaces, and components.
- A design should lead to data structures that are appropriate for the classes to be implemented and are drawn from recognizable data patterns.
- A design should lead to components that exhibit independent functional characteristics

Quality Guidelines (contd.)

- A design should lead to interfaces that reduce the complexity of connections between components and with the external environment.
- A design should be derived using a repeatable method that is driven by information obtained during software requirements analysis.
- A design should be represented using a notation that effectively communicates its meaning.

Design Principles

- S/W design is both a process and a model.
- *Design process* - sequence of steps that enable the designer to describe all aspects of the software to be built.
- *Design model* - created for software provides a variety of different views of the computer software

- *The design process* *<u>should not suffer from ‘tunnel vision</u>.’* - Designer should consider alternative approaches.
- *The design should be* *<u>traceable to the analysis model</u>* - a single element of the design model often traces to multiple requirements, it is necessary to have a means for tracking how requirements have been satisfied by the design model.
- *The design should* *<u>not reinvent the wheel</u>*- use already exists design pattern because time is short and resource are limited.
- *The design should “<u>minimize the intellectual distance</u>” between the software and the problem as it exists in the real world.* – design should be self-explanatory

- *The design should <u>exhibit uniformity and integration</u>* – before design work begins rules of styles and format should be defined for a design team.
- *The design should be <u>structured to accommodate change</u>*
- *The design should be structured to <u>degrade gently</u>, even when unusual 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.*

Design concepts

- Design concepts provide the necessary framework for “to get the thing on right way”.
- Abstraction
- Refinement
- Modularity
- Architecture
- Information Hiding
- Patterns
- Separation of concerns
- Functional Independence
- Aspects
- Refactoring

Abstraction

- At the highest level of abstraction – a solution is stated in broad terms
- At lower level of abstraction – a more detailed description of the solution is provided.
- Two types of abstraction:
- *Procedural abstraction:* Sequence of instructions that have a specific and limited function.

Ex. Open a door

open implies long sequence of activities (e.g. walk to the door, grasp knob, turn knob and pull the door, etc).

- *Data abstraction*: collection of data that describes a data object.

Ex. Open a door. – **door** is data object.

- Data abstraction for ***door*** would encompass a set of attributes that describe the door. (E.g. door type, swing direction, opening mechanism, etc.)

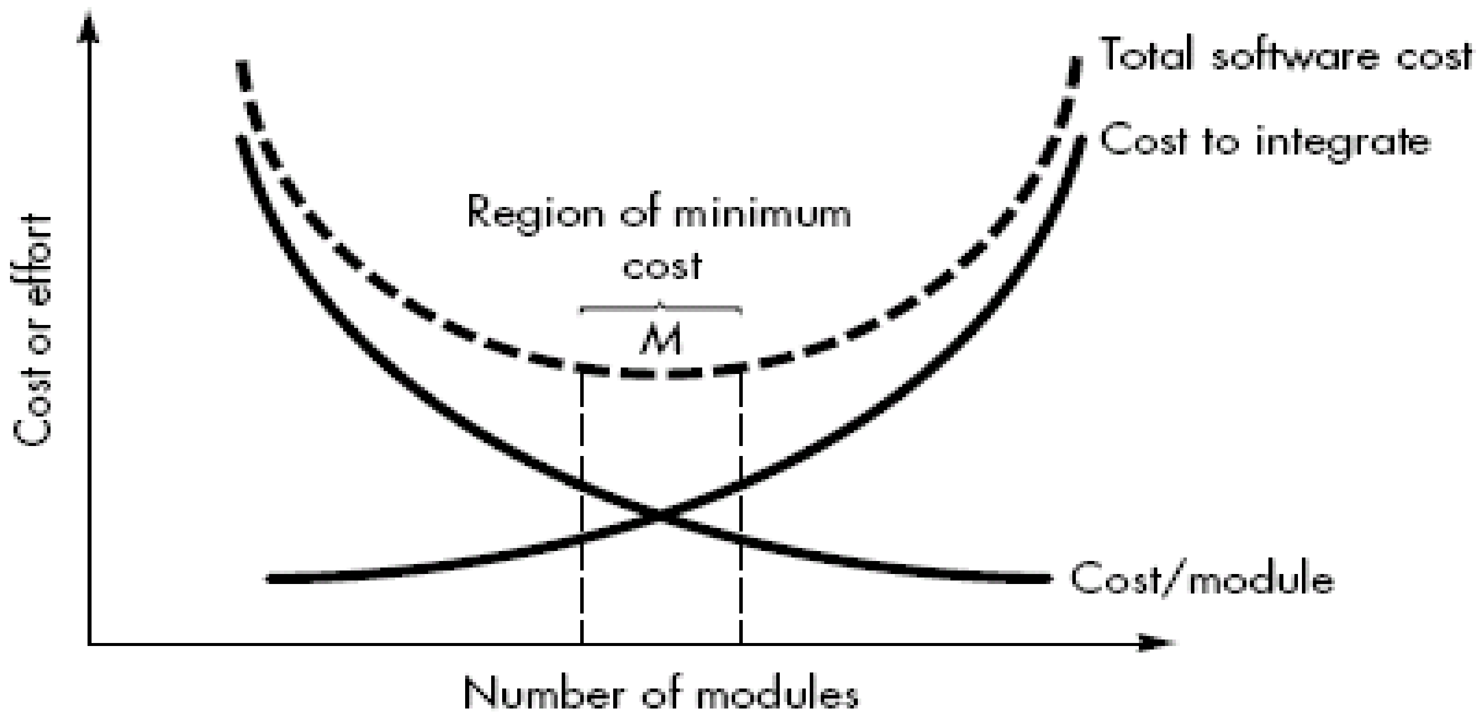
Refinement

- Refinement is actually a process of *elaboration.*
- begin with a statement of function (or description of information) that is defined at a high level of abstraction.
- That is, the statement describes function or information conceptually but provides no information about the internal workings of the function or the internal structure of the information.
- Refinement causes the designer to elaborate on the original statement, providing more and more detail as each successive refinement (elaboration) occurs.
- Abstraction and refinement are complementary concepts. Abstraction enables a designer to specify procedure and data and yet suppress low-level details.
- Refinement helps the designer to expose low-level details as design progresses.

Modularity

- Architecture and design pattern embody modularity.
- Software is divided into separately named and addressable components, sometimes called modules, which are integrated to satisfy problem requirement.
- modularity is the single attribute of software that allows a program to be intellectually manageable
- It leads to a “divide and conquer” strategy. – it is easier to solve a complex problem when you break into a manageable pieces.
- Refer fig. that state that effort (cost) to develop an individual software module does decrease if total number of modules increase.
- However as the no. of modules grows, the effort (cost) associated with integrating the modules also grows.

Modularity and software cost



- Undermodularity and overmodularity should be avoided. But how do we know the vicinity of M?
- We modularize a design so that development can be more easily planned.
- Software increments can be defined and delivered.
- Changes can be more easily accommodated.
- Testing and debugging can be conducted more efficiently and long-term maintained can be conducted without serious side effects.

Architecture

- Software architecture suggest “ the overall structure of the software and the ways in which that structure provides conceptual integrity for a system.
- No. of different models can use to represent architecture.
  - Structural Model- represent architecture as an organized collection of components
  - Framework model – Increase level of design abstraction by identifying repeatable architectural design framework.
  - Dynamic model – address behavior of the program architecture and indicating how system or structure configuration may change as a function.
  - Process Model – focus on design of the business or technical process that the system must accommodate.
  - Functional models – used to represent the functional hierarchy of a system.

Information Hiding

- The principle of *information hiding* suggests that modules be "characterized by design decisions that (each) hides from all others modules.“
- In other words, modules should be specified and designed so that information (algorithm and data) contained within a module is inaccessible to other modules that have no need for such information.
- The intent of information hiding is to hide the details of data structure and procedural processing behind a module interface.
- It gives benefits when modifications are required during testing and maintenance because data and procedure are hiding from other parts of software, unintentional errors introduced during modification are less.

Separation of concerns

- It suggests that any complex problem can be more easily handled if it is sub-divided into pieces that can each be solved or optimized independently.

Refactoring

- It is a reorganization technique that simplifies the design (or code) of acomponent without changing its function or behaviour.
- It is a process of changing a software system in such a way that it doesn’t alter the external behaviour of the code(design) yet improve its internal structure

Aspects

- An aspect is a representation of crosscutting concern.
- Ex: Only valid user can access the ATM.

- It is important to identify aspects that the design an properly accommodate them as refinement and modularization occur.

Patterns

- A design pattern describes a design structure that solves a particular design problem within a specific context that may have an impact on the manner in which the pattern is applied and used.

  1.Whether the pattern is applicable to the current work

  2. Whether the pattern can be reused

  3. Whether the pattern ca serve as a guide for developing a similar, but functionally or structurally different pattern.

EFFECTIVE MODULAR DESIGN

- Effective modular design consist of three things:
  - Functional Independence
  - Cohesion
  - Coupling
  - Fan-out
  - Fan-in
  - Factoring

Functional Independence

- Functional independence is achieved by developing modules with "single-minded“ function and an "aversion" to excessive interaction with other modules.
- In other words - each module addresses a specific sub-function of requirements and has a simple interface when viewed from other parts of the program structure.
- Independence is important –
  - Easier to develop
  - Easier to Test and maintain
  - Error propagation is reduced
  - Reusable module.

Functional Independence

- To summarize, functional independence is a key to good design, and design is the key to software quality.
- To measure independence, have two qualitative criteria: cohesion and coupling
- *Cohesion* is a measure of the relative functional strength of a module.
- *Coupling* is a measure of the relative interdependence among modules.

Cohesion

- Cohesion is a natural extension of the information hiding concept
- A cohesive module performs a single task within a software procedure, requiring little interaction with procedures being performed in other parts of a program
- Simply state, a cohesive module should (ideally) do just one thing.
- We always <u>strive for high cohesion</u>, although the <u>mid-range </u>of the spectrum is often <u>acceptable.</u>
- Low-end cohesiveness is much "worse" than middle range, which is nearly as "good" as high-end cohesion.
- So. designer should avoid low levels of cohesion when modules are designed.

Cohesion

- When processing elements of a module are related and must be executed in a specific order, *procedural cohesion* exists.
- When all processing elements concentrate on one area of a data structure, *communicational cohesion* is present.
- High cohesion is characterized by a module that performs one distinct procedural task.

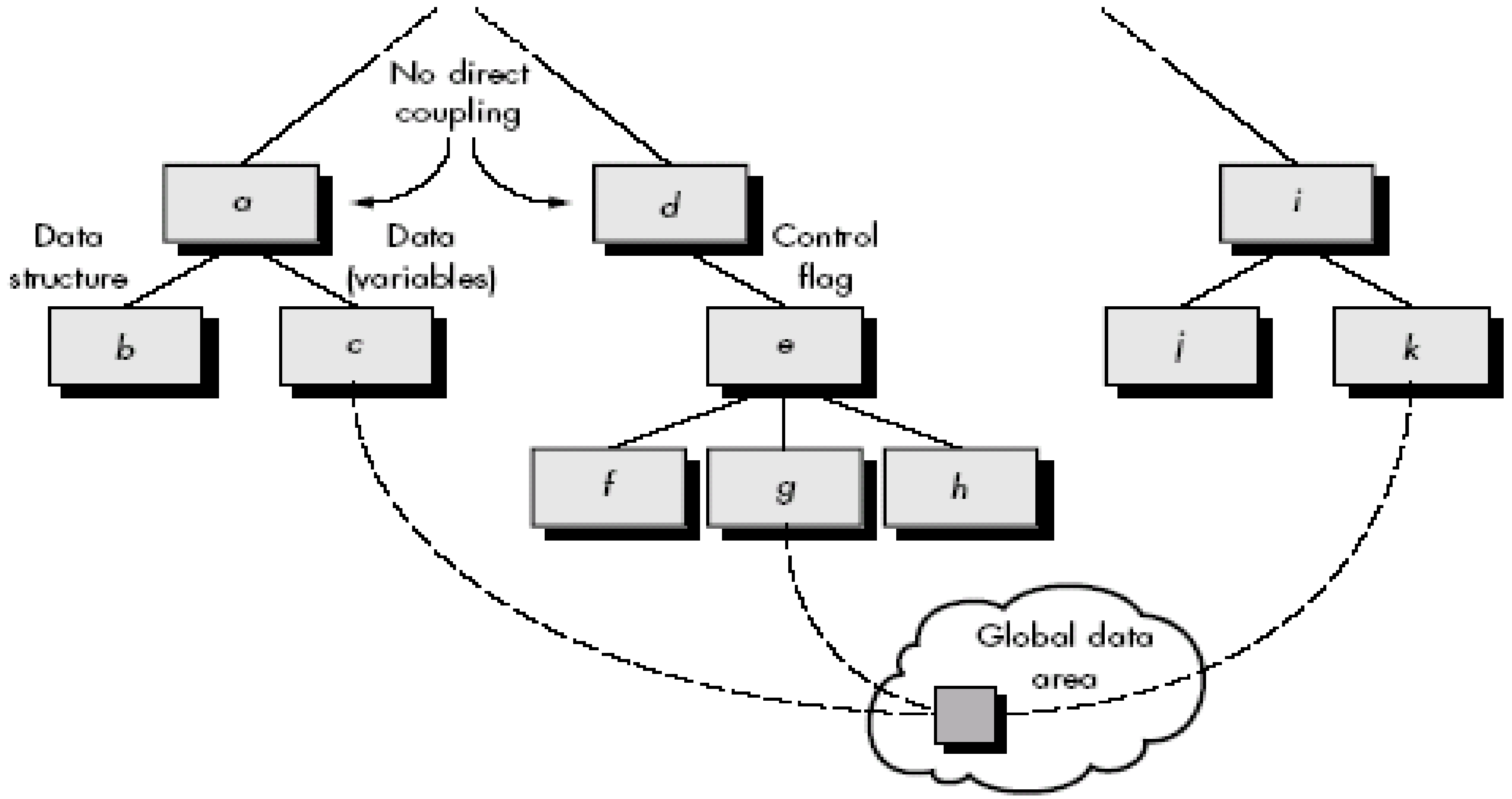
Types of cohesion

- A module that performs tasks that are related logically is *logically cohesive*.
- When a module contains tasks that are related by the fact that all must be executed with the same span of time, the module exhibits *temporal cohesion*.
- At the low-end of the spectrum, a module that performs a set of tasks that relate to each other loosely, called *coincidentally cohesive.*

Coupling

- Coupling 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
- In software design, we strive for lowest possible coupling. Simple connectivity among modules results in software that is easier to understand and less prone to a "ripple effect" caused when errors occur at one location and propagate through a system.
- It occur because of design decisions made when structure was developed.

Coupling


No direct coupling
a
d
i
Data structure
Data (variables)
Control flag
b
c
e
j
k
f
g
h
Global data area

Coupling

- Coupling is characterized by passage of control between modules.
- “Control flag” (a variable that controls decisions in a subordinate or superordinate module) is passed between modules d and e (called control coupling).
- Relatively high levels of coupling occur when modules are communicate with external to software.
- External coupling is essential, but should be limited to a small number of modules with a structure.

- High coupling also occurs when a number of modules reference a global data area.
- Common coupling, no. of modules access a data item in a global data area
- So it does not mean “use of global data is bad”. It does mean that a software designer must be take care of this thing.

Evolution of Software Design

- Early design work concentrated on criteria for the development of modular programs and methods for refining software structures in a top-down manner.
- Later work proposed methods for the translation of data flow or data structure into a design definition.
- Today, the emphasis in software design has been on software architecture and the design patterns that can be used to implement software architectures.

Characteristics are common to all design methods

- A mechanism for the translation of analysis model into a design representation,
- A notation for representing functional components and their interfaces.
- Heuristics for refinement and partitioning
- Guidelines for quality assessment.

Design quality attributes

- Acronym FURPS –
  - Functionality
  - Usability
  - Reliability
  - Performance
  - Supportability

- Functionality – is assessed by evaluating the feature set and capabilities of the program.
  - Functions that are delivered and security of the overall system.
- Usability – assessed by considering human factors, consistency & documentation.
- Reliability – evaluated by
  - measuring the frequency and severity of failure.
  - Accuracy of output results.
  - Ability to recover from failure and predictability of the program.
- Performance - measured by processing speed, response time, resource consumption, efficiency.
- Supportability – combines the ability to extend the program (extensibility), adaptability and serviceability.

DESIGN HEURISTICS

- Evaluate 1st iteration to reduce coupling & improve cohesion
- Minimize structures with high fan-out; strive for depth
- Keep scope of effect of a module within scope of control of that module
- Evaluate interfaces to reduce complexity and improve consistency

Continued…

- Define modules with predictable function & avoid being overly restrictive
  - Avoid static memory between calls where possible
- Strive for controlled entry -- no jumps into the middle of things
- Package software based on design constraints and portability requirements

Software Architecture

- "The software architecture of a program or computing system is the structure or structures of the system, which comprise the software components, the externally visible properties of those components, and the relationships among them."
- It is not operational software but it is representation that enables software engineer to
  - Analyze the effectiveness of design in meeting its stated requirement.
  - consider architectural alternatives at a stage when making design changes is still relatively easy,
  - reduce the risks associated with the construction of the software.

Importance of Software Architecture

- Representations of software architecture are an enabler for communication between all parties (stakeholders) interested in the development
- Architecture highlights early design decisions that will have a profound impact on all software engineering work that follows.
- Architecture “constitutes a relatively small, intellectually graspable model of how the system is structured and how its components work together”

Architectural Style

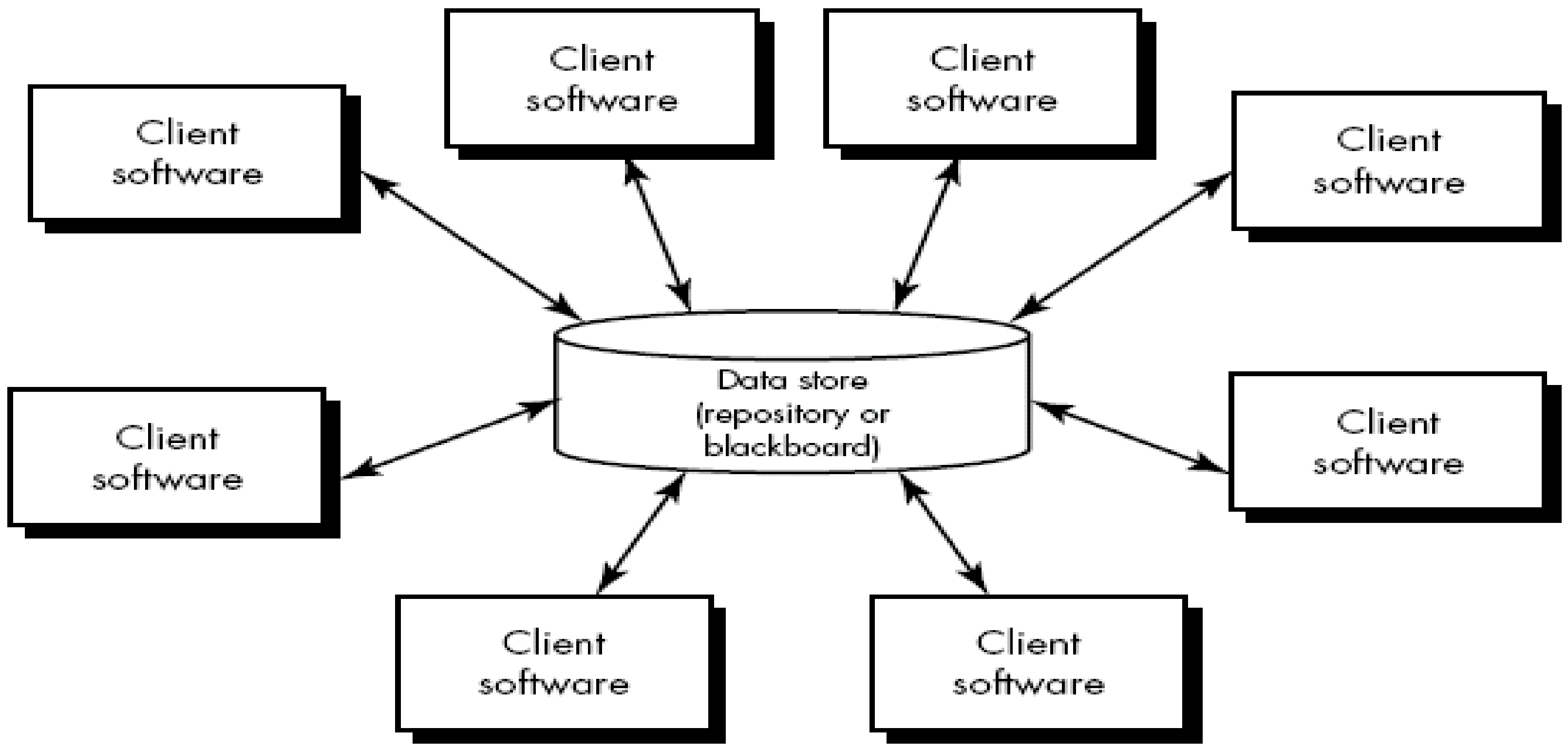
- Style describes a system category that encompasses

1. A set of *components* (e.g., a database, computational modules) that perform a function required by a system;
2. a set of *connectors* that enable “communication, co-ordinations and cooperation” among components;
3. *constraints* that define how components can be integrated to form the system
4. *semantic models* that enable a designer to understand the overall properties of a system

It can be represent by

- Data-centered architecture
- Data flow architecture
- Call and return architecture
- Object oriented architecture
- Layered architecture.

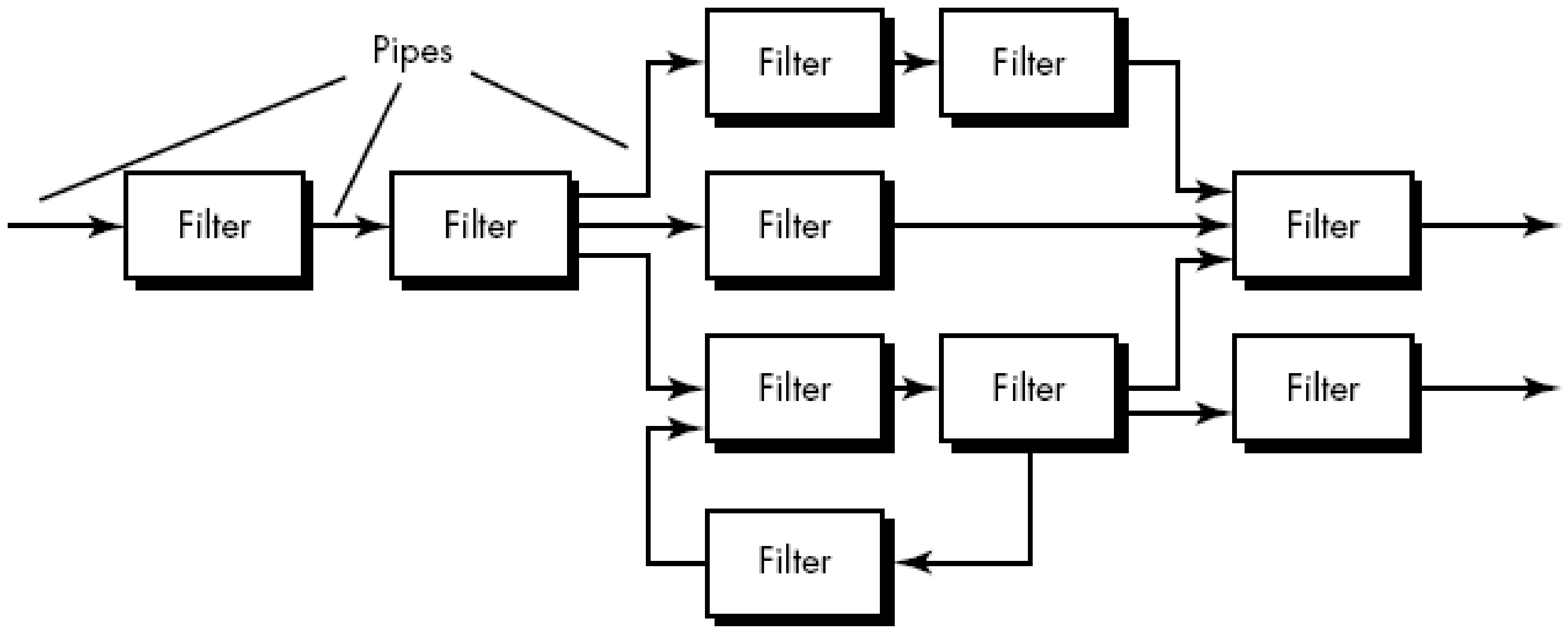
Data-centered architecture



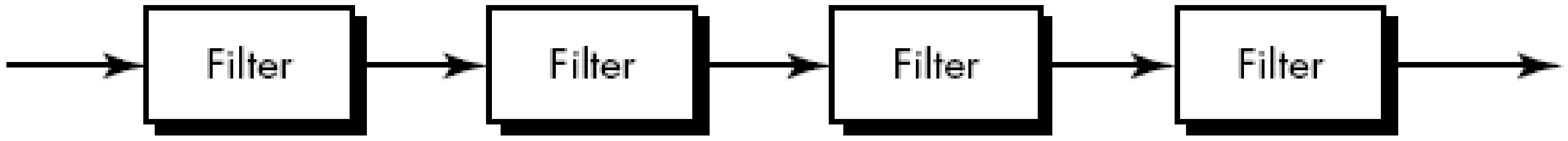
Data-centered architecture

- A data store (e.g., a file or database) resides at the center of this architecture and is accessed frequently by other components that update, add, delete, or otherwise modify data within the store.
- Client software accesses a central repository which is in passive state (in some cases).
- client software accesses the data independent of any changes to the data or the actions of other client software.
- So, in this case transform the repository into a “Blackboard”.
- A blackboard sends notification to subscribers when data of interest changes, and is thus active.
- Data-centered architectures promote *integrability.*
- Existing components can be changed and new client components can be added to the architecture without concern about other clients.
- Data can be passed among clients using the blackboard mechanism. So Client components independently execute processes

Data Flow architecture



Pipes and filters

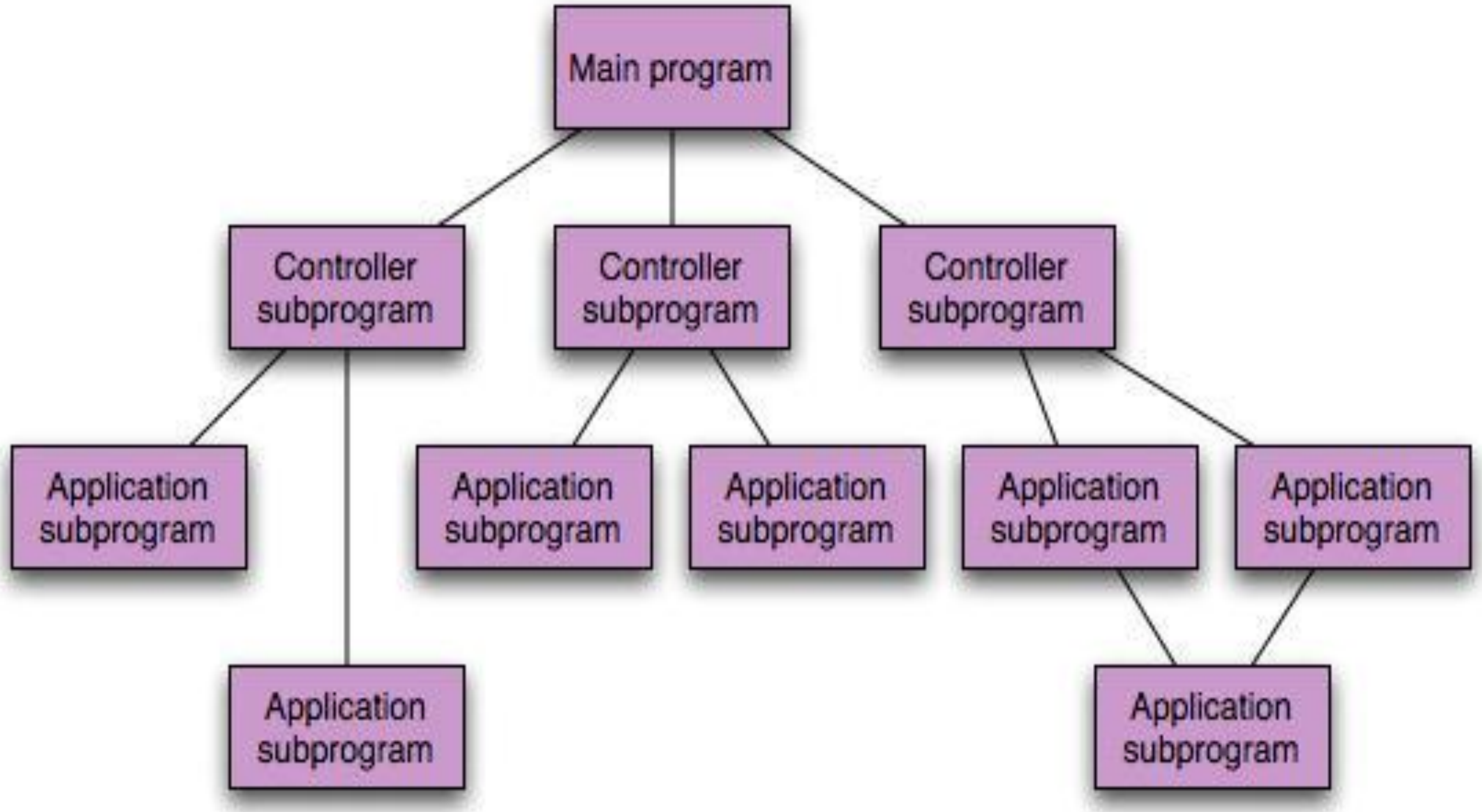


Batch Sequential

Data Flow architecture

- This architecture is applied when input data are to be transformed through a series of computational or manipulative components into output data.
- A *pipe and filter pattern (Fig .1)* has a set of components, called *filters,* connected by pipes that transmit data from one component to the next.
- Each filter works independently (i.e. upstream, downstream) and is designed to expect data input of a certain form, and produces data output (to the next filter) of a specified form.
- the filter does not require knowledge of the working of its neighboring filters.
- If the data flow degenerates into a single line of transforms, it is termed *batch sequential.*

Call and return architecture



Call and return architecture

- Architecture style enables a software designer (system architect) to achieve a program structure that is relatively easy to modify and scale.
- Two sub-styles exist within this category:

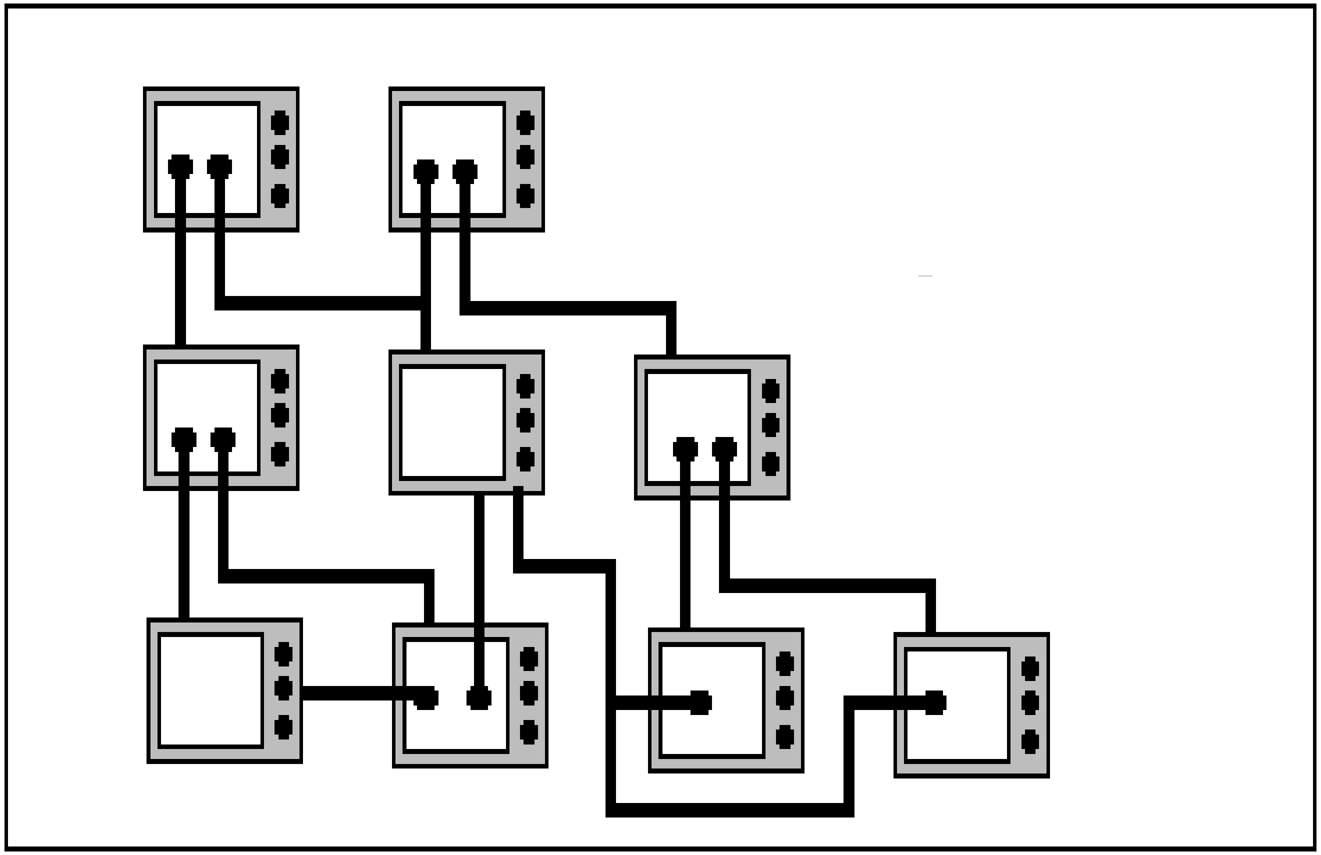
1. *Main/sub program architecture:*

- Program structure decomposes function into a control hierarchy where a “main” program invokes a number of program components, which in turn may invoke still other components.

2. *Remote procedure Call architecture:*

- The components of a main program/subprogram architecture are distributed across multiple computers on a network

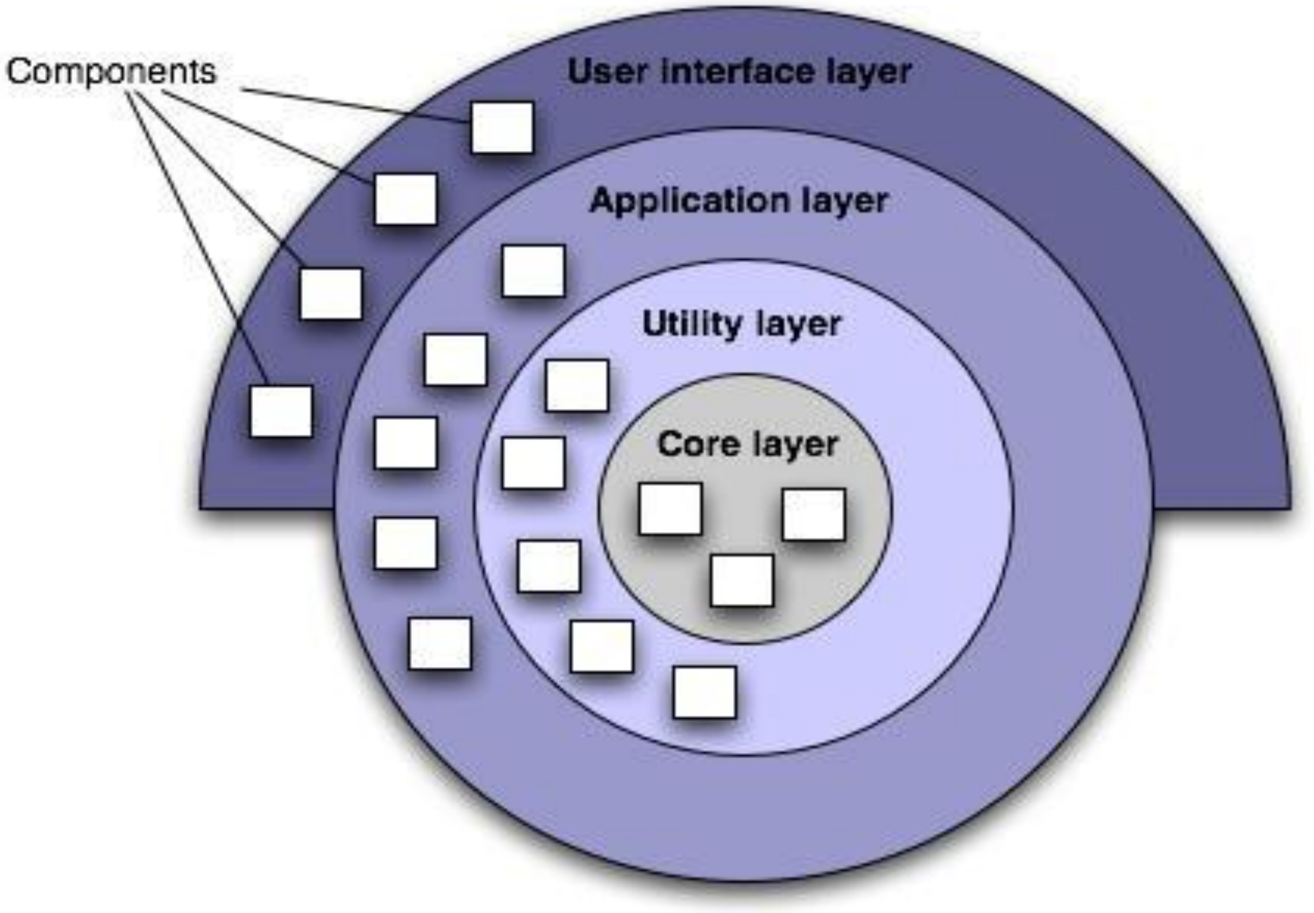
Object-oriented architecture



Object-oriented architecture

- The object-oriented paradigm, like the abstract data type paradigm from which it evolved, emphasizes the bundling of data and methods to manipulate and access that data (Public Interface).
- Components of a system summarize data and the operations that must be applied to manipulate the data.
- Communication and coordination between components is accomplished via message passing.

Layered Architecture

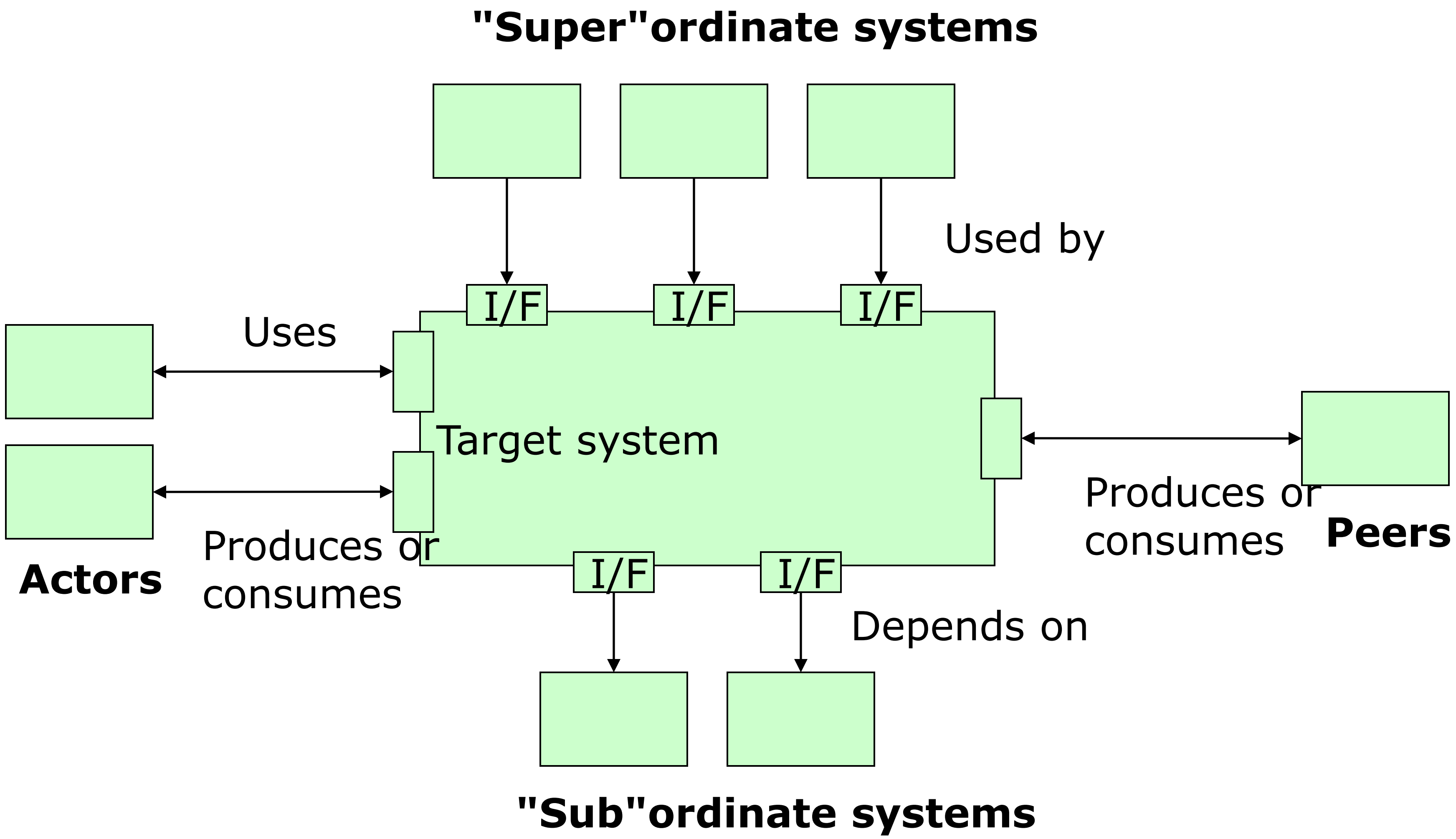


Architectural Design Steps

1) Represent the system in context
2) Define archetypes
3) Refine the architecture into components
4) Describe instantiations of the system

"A doctor can bury his mistakes, but an architect can only advise his client to plant vines." Frank Lloyd Wright

1. Represent the System in Context

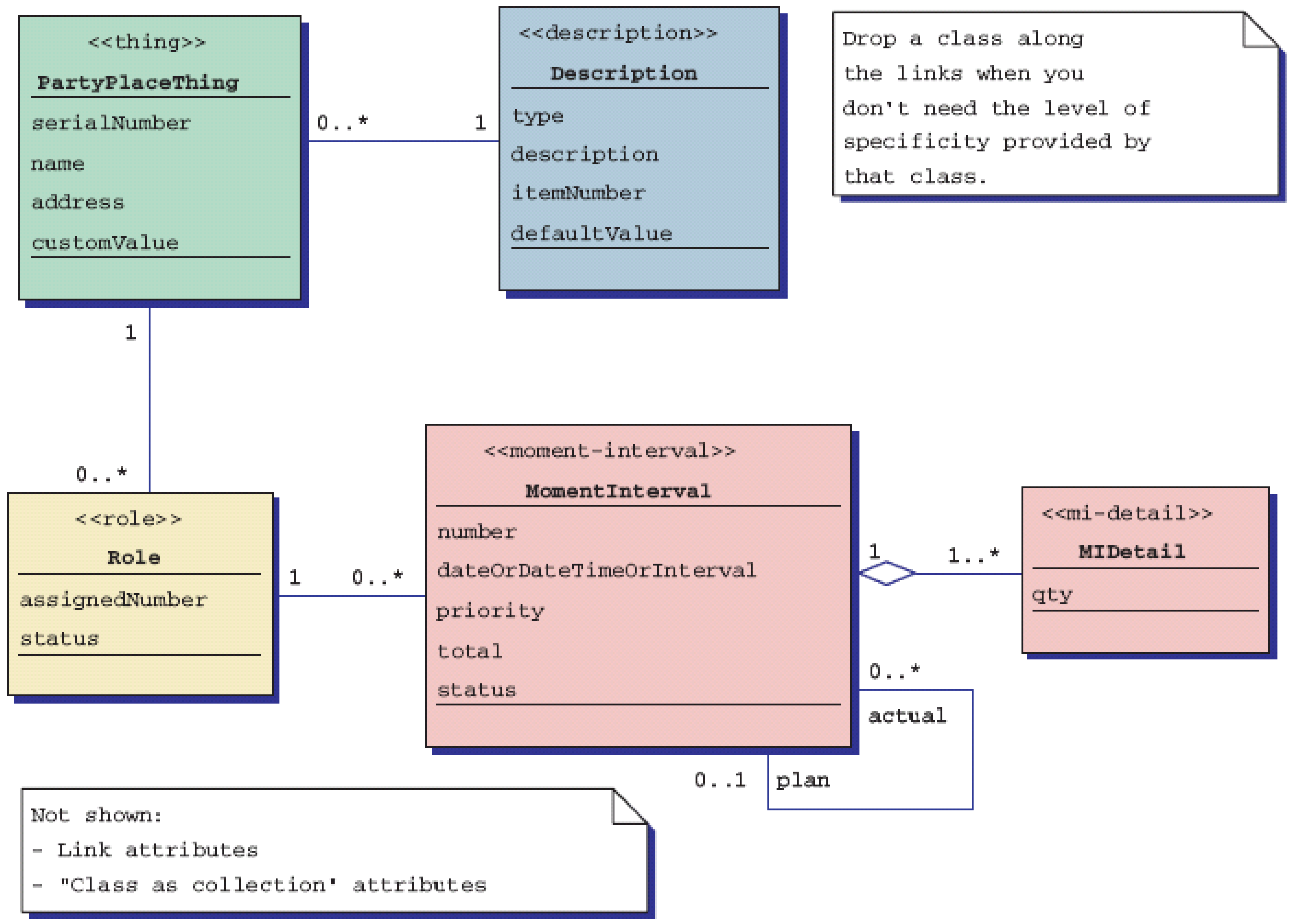


(More on next slide)

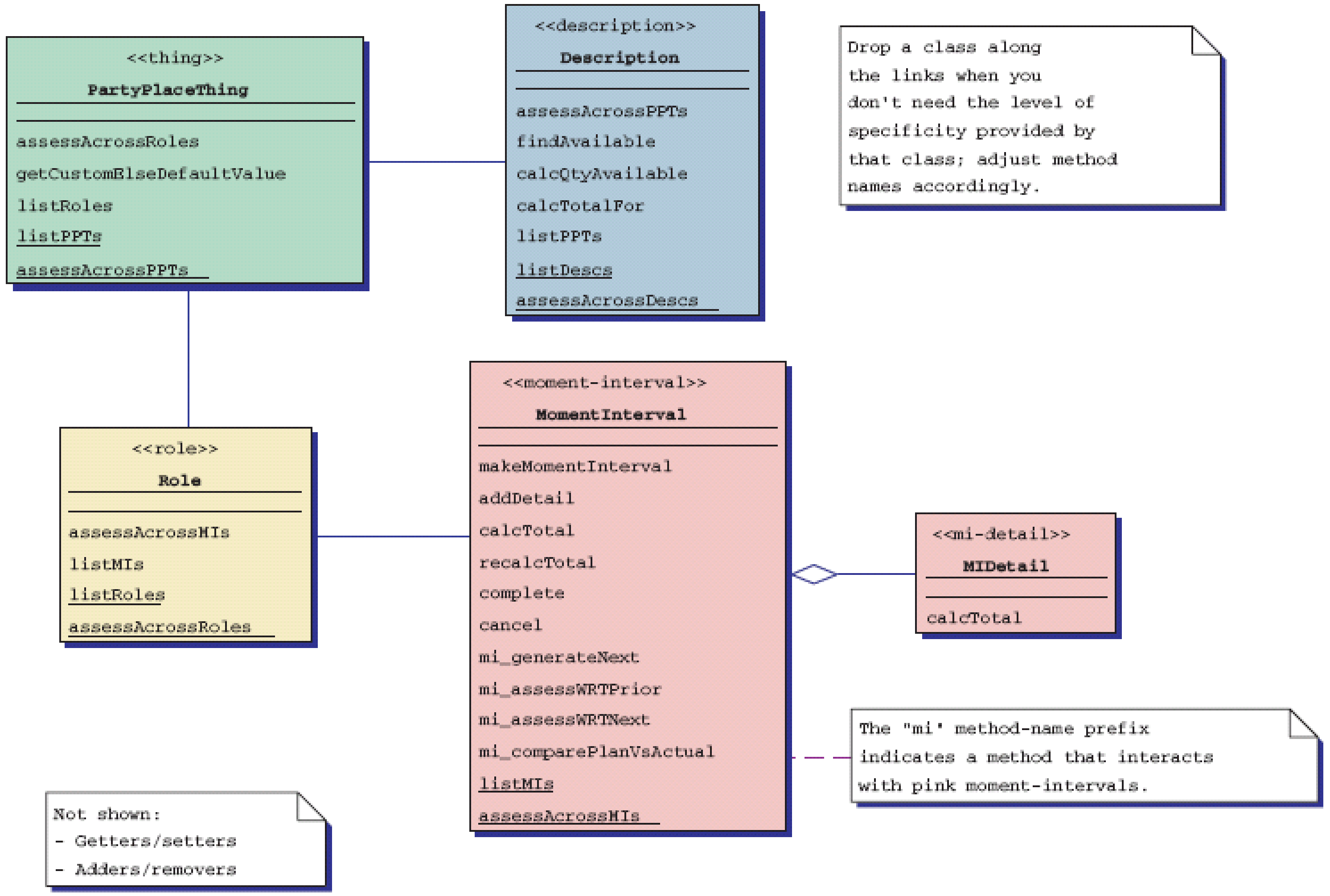
1. Represent the System in Context (continued)

- Use an architectural context diagram (ACD) that shows
  - The <u>identification</u> and <u>flow</u> of all information into and out of a system
  - The specification of all <u>interfaces</u>
  - Any relevant <u>support processing</u> from/by other systems
- An ACD models the manner in which software interacts with entities <u>external</u> to its boundaries
- An ACD identifies systems that interoperate with the target system
  - Super-ordinate systems
    - Use target system as part of some higher level processing scheme
  - Sub-ordinate systems
    - Used by target system and provide necessary data or processing
  - Peer-level systems
    - Interact on a peer-to-peer basis with target system to produce or consume data
  - Actors
    - People or devices that interact with target system to produce or consume data

Archetypes – their attributes



Archetypes – their methods



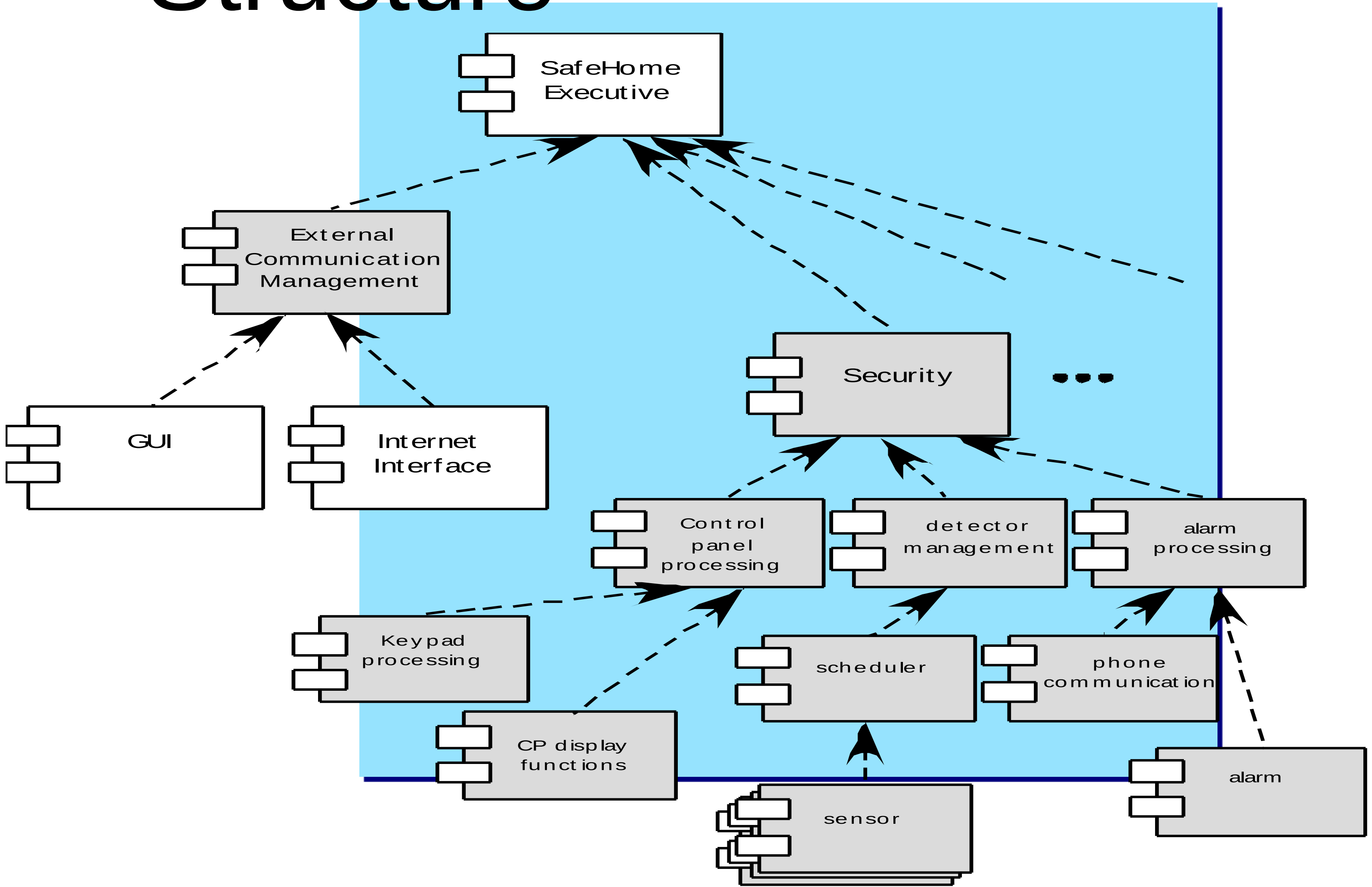
3. Refine the Architecture into Components

- Based on the archetypes, the architectural designer refines the software architecture into components to illustrate the overall structure and architectural style of the system
- These components are derived from various sources
  - The application domain provides application components, which are the domain classes in the analysis model that represent entities in the real world
  - The infrastructure domain provides design components (i.e., design classes) that enable application components but have no business connection
    - Examples: memory management, communication, database, and task management
  - The interfaces in the ACD imply one or more specialized components that process the data that flow across the interface
- A UML class diagram can represent the classes of the refined architecture and their relationships

4. Describe Instantiations of the System

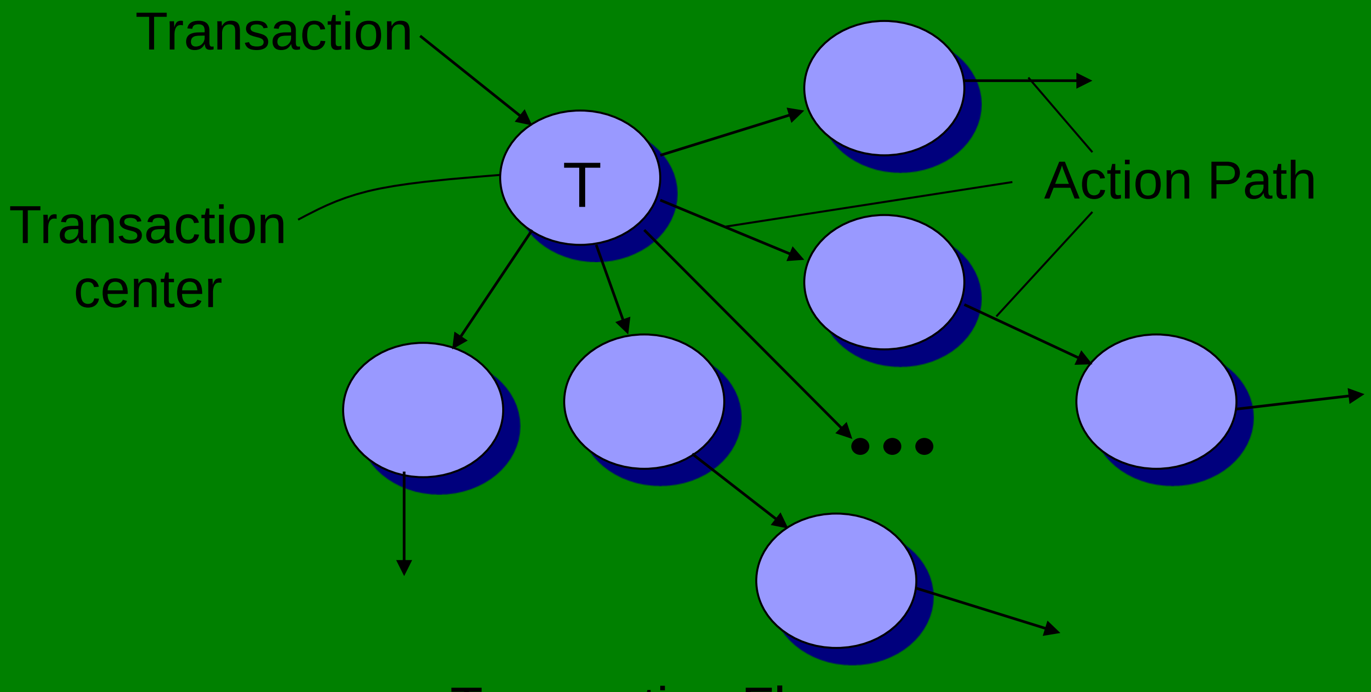
- An actual instantiation of the architecture is developed by applying it to a specific problem
- This demonstrates that the architectural structure, style and components are appropriate
- A UML component diagram can be used to represent this instantiation

Refined Component Structure

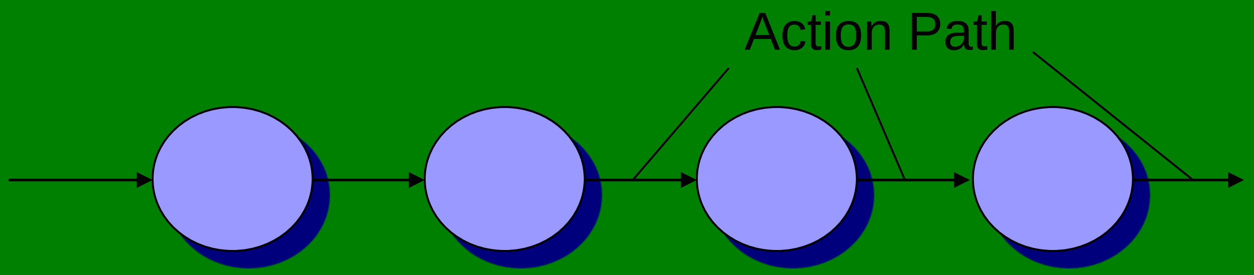


Mapping Requirements to Software Architecture

- Establish type of information flow
  - transform flow - overall data flow is sequential and flows along a small number of straight line paths
  - transaction flow - a single data item triggers information flow along one of many paths

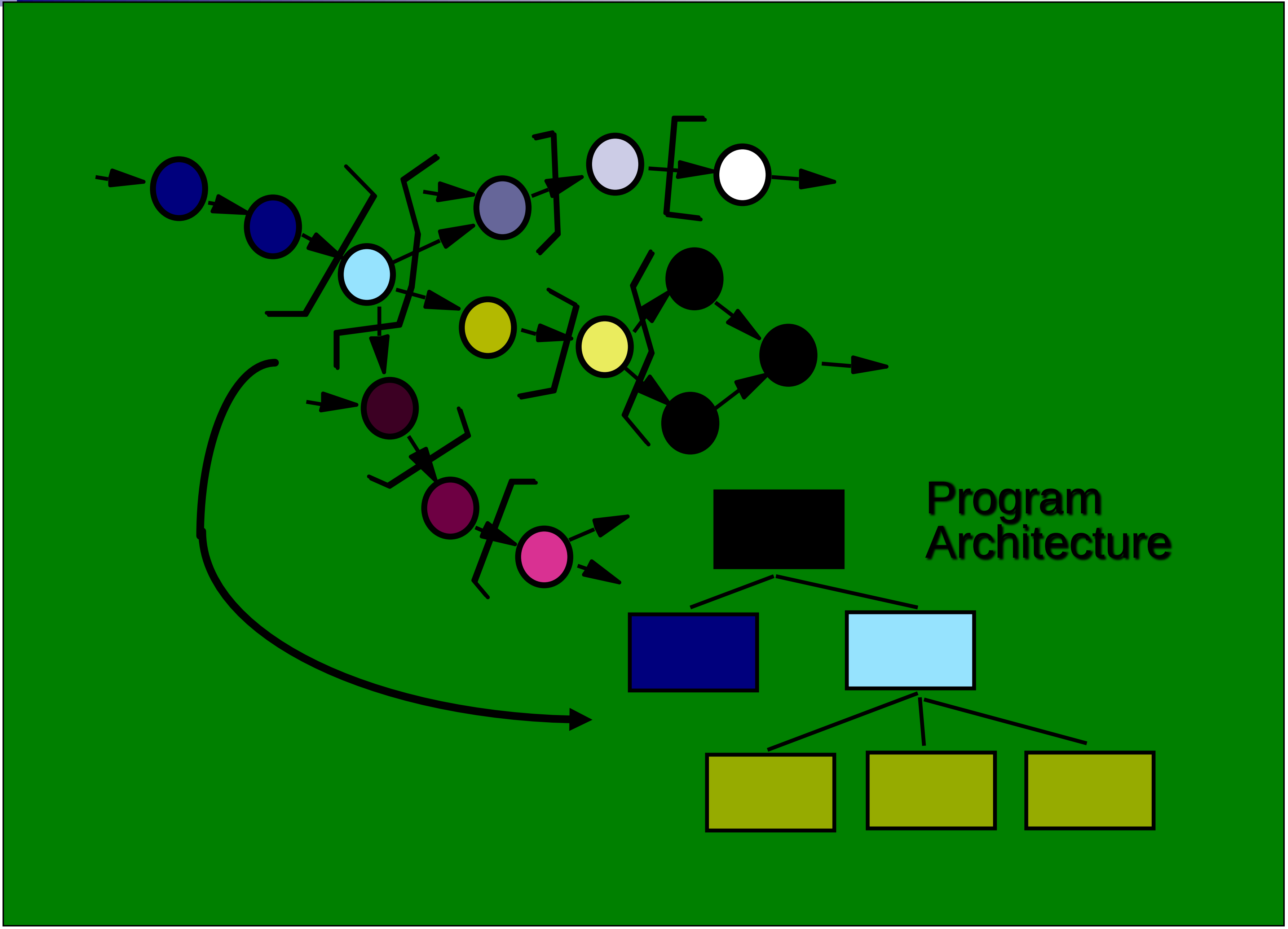


Transaction Flow



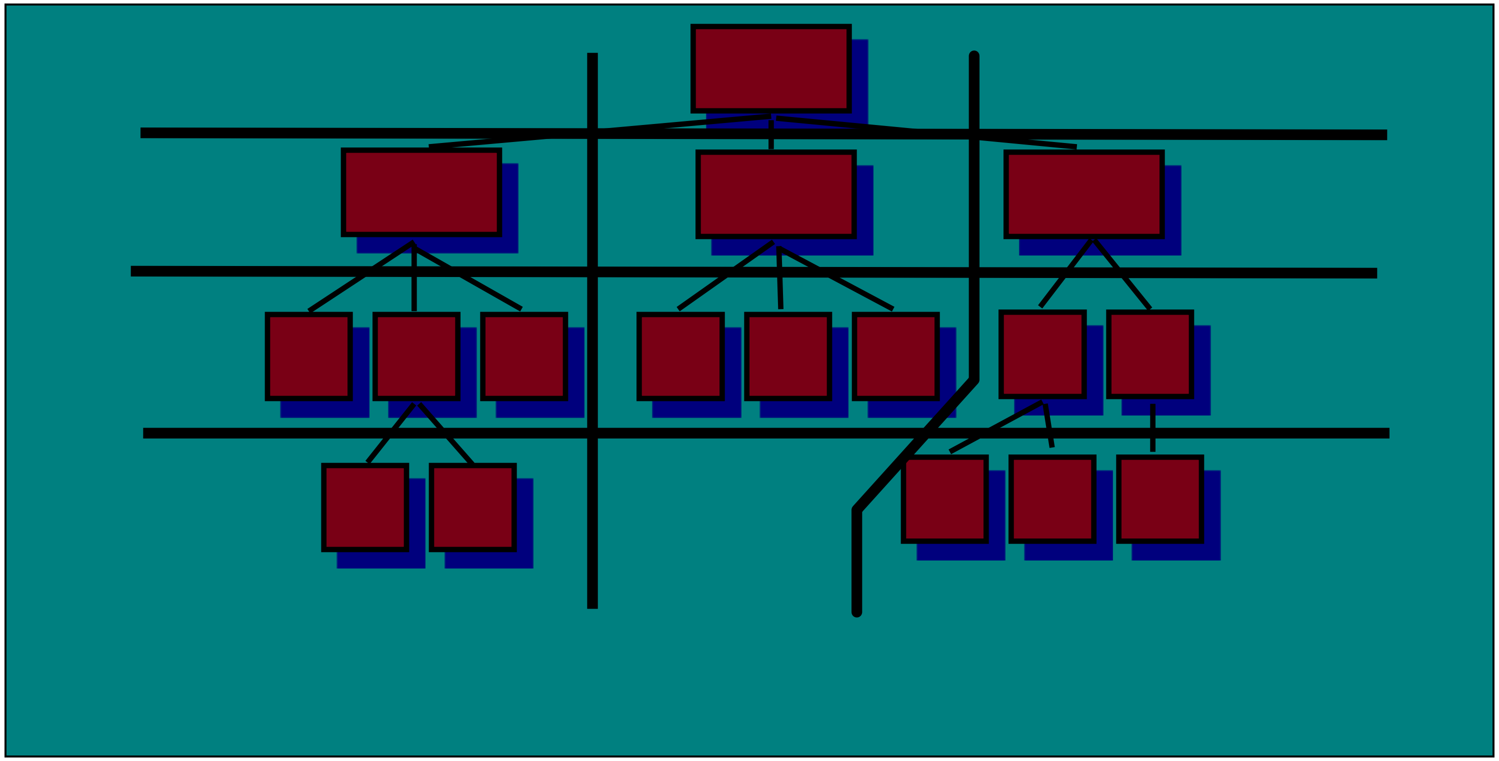
Transform Flow

- Flow boundaries indicated
- DFD is mapped into program structure
- Control hierarchy defined
- Resultant structure refined using design measures and heuristics
- Architectural description refined and elaborated


Program
Architecture

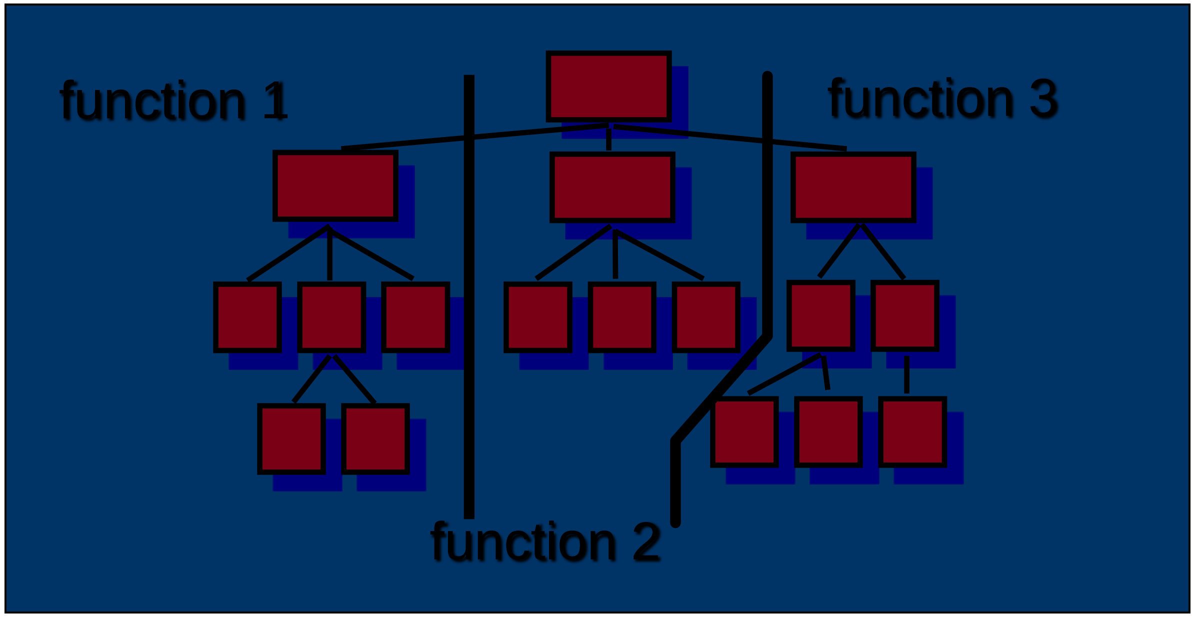
•Partitioning the Architecture

- “horizontal” and “vertical” partitioning are required



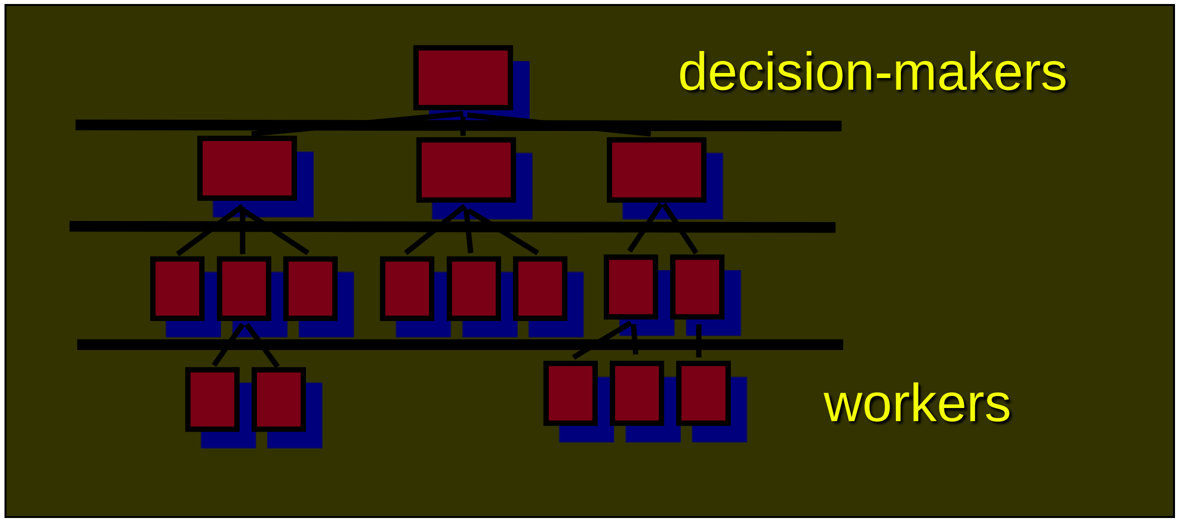
•Horizontal Partitioning

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



•Vertical Partitioning: Factoring

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



Why Partitioned Architecture?

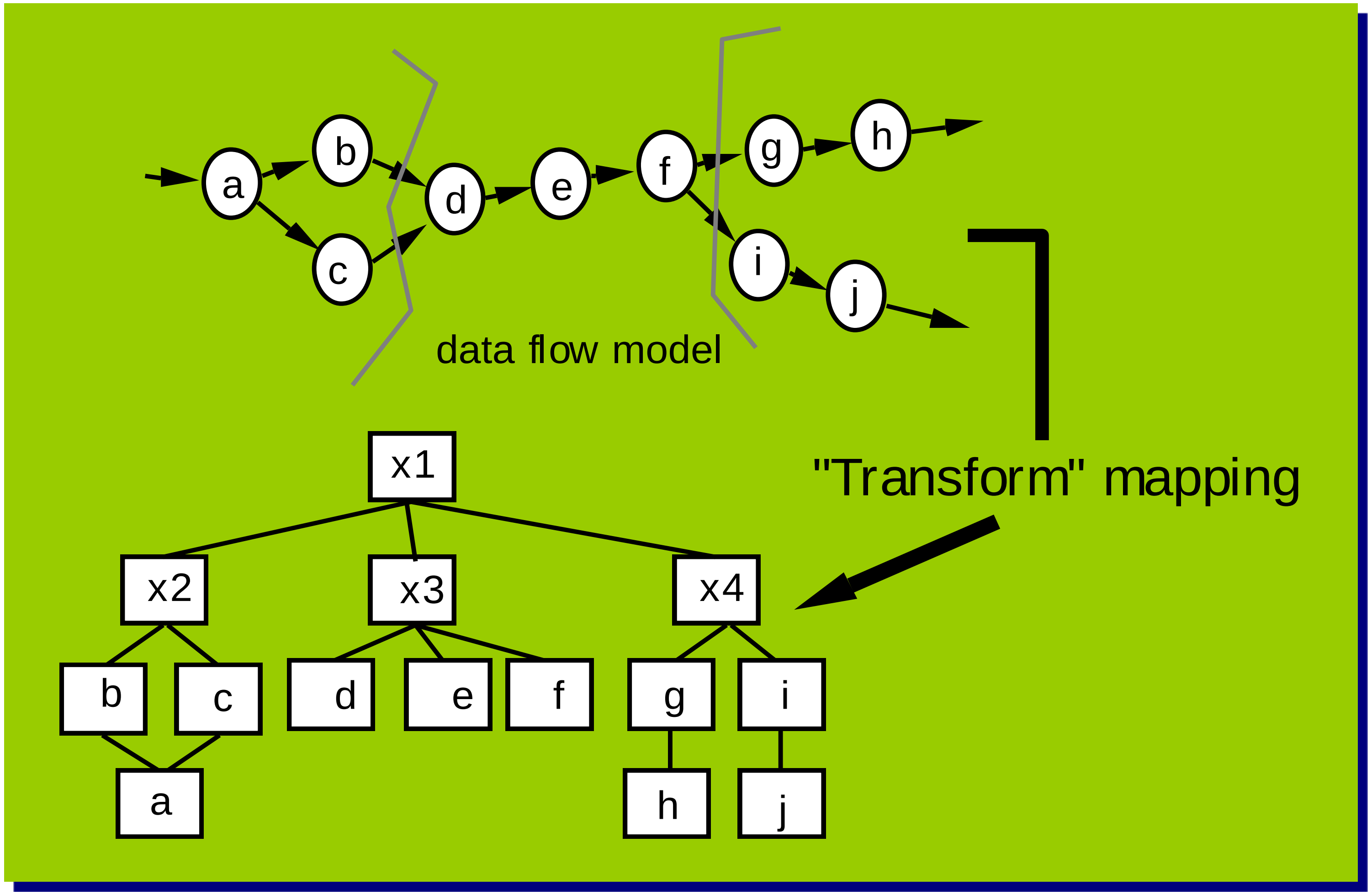
- results in software that is easier to test
- leads to software that is easier to maintain
- results in propagation of fewer side effects
- results in software that is easier to extend

Transform Mapping

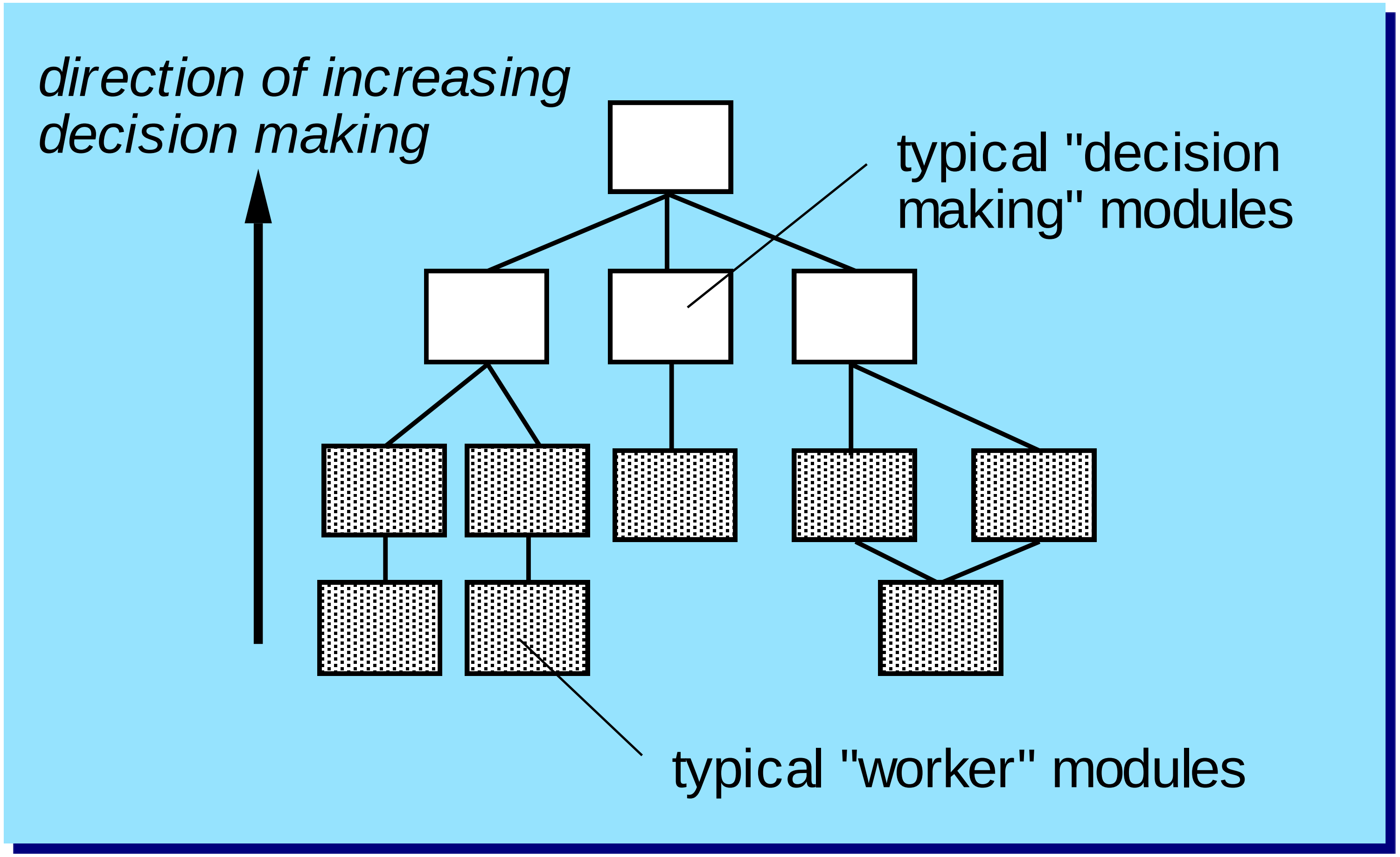
- Review fundamental system model
- Review and refine data flow diagrams for the software
- Determine whether the DFD has transform or transaction characteristics

- Isolate the transform center by specifying incoming and outgoing flow boundaries
- Perform first level factoring
- Perform second level factoring
- Refine the first iteration architecture using design heuristics for improved software quality

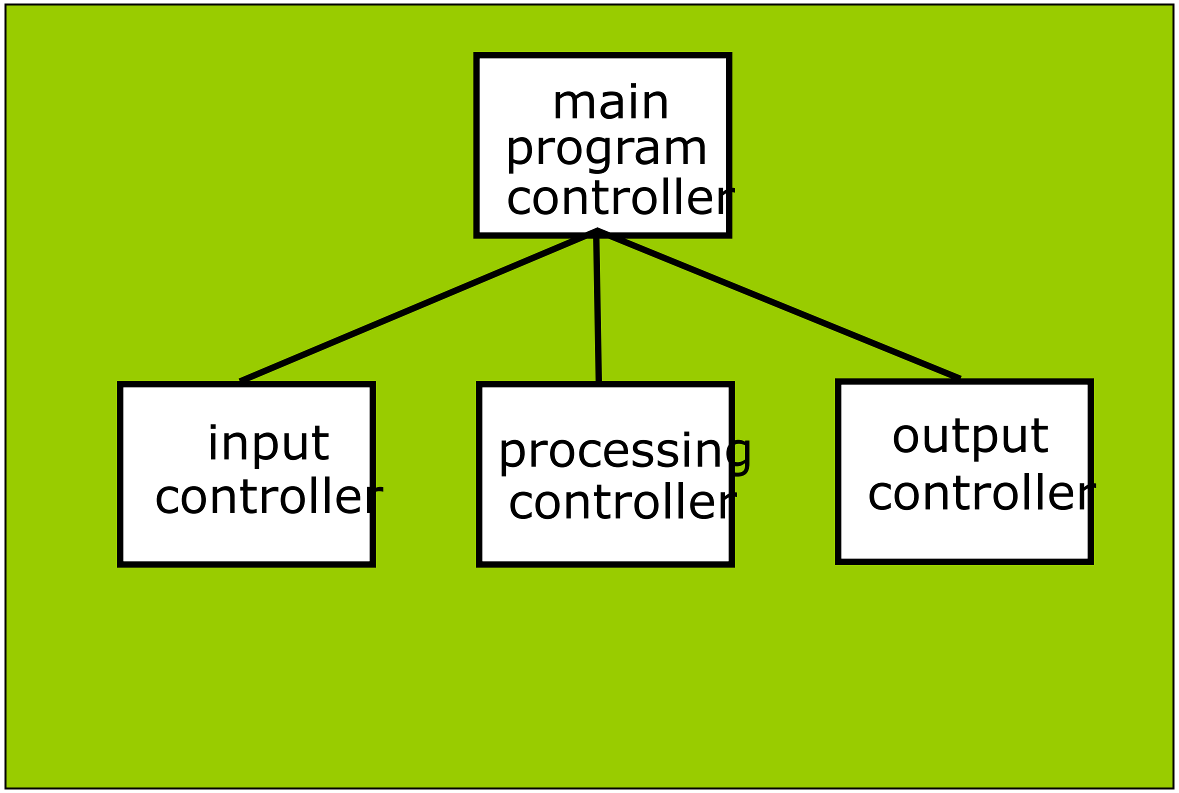
Transform Mapping



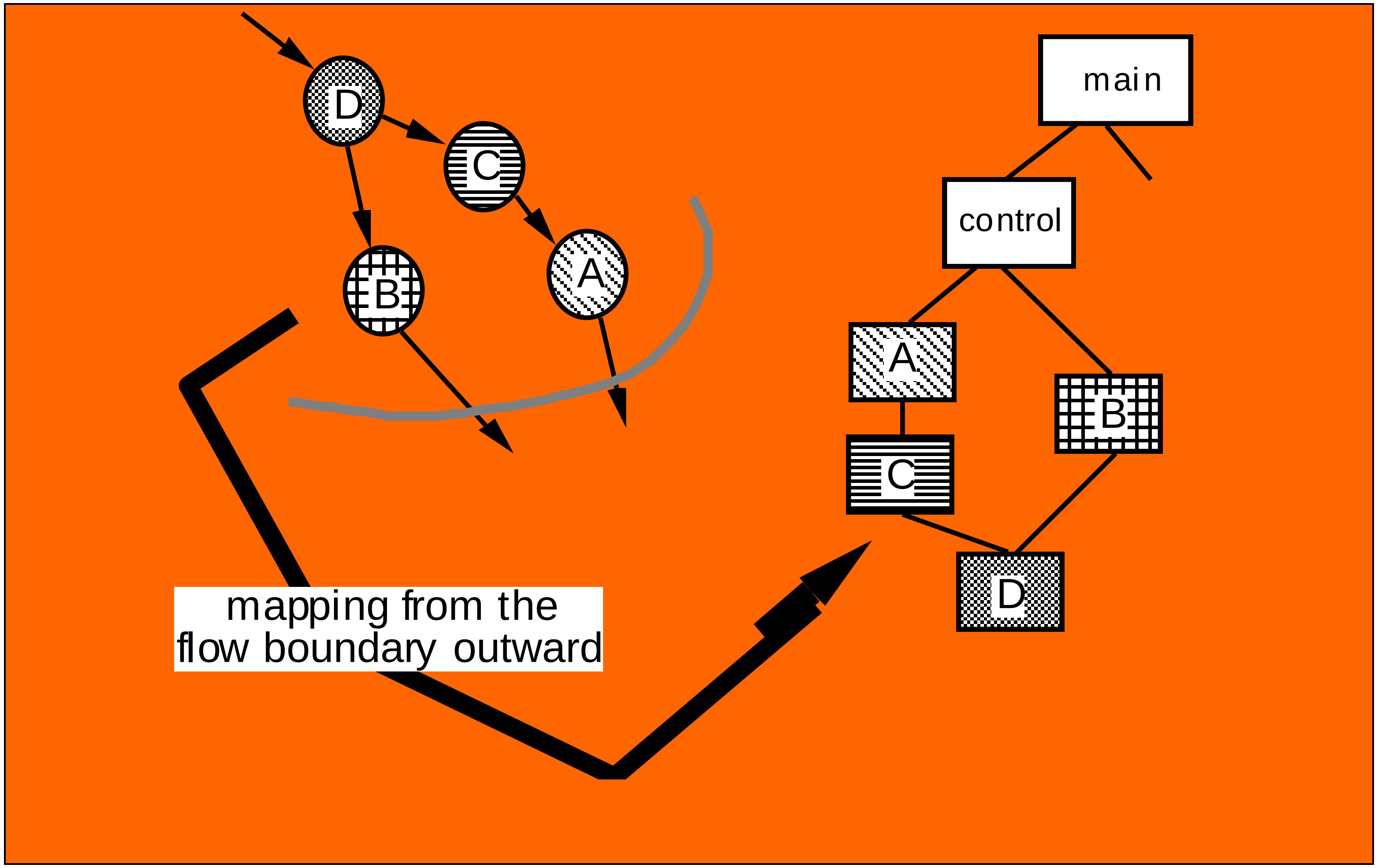
Factoring


direction of increasing decision making
typical "decision making" modules
typical "worker" modules

First Level Factoring



Second Level Mapping

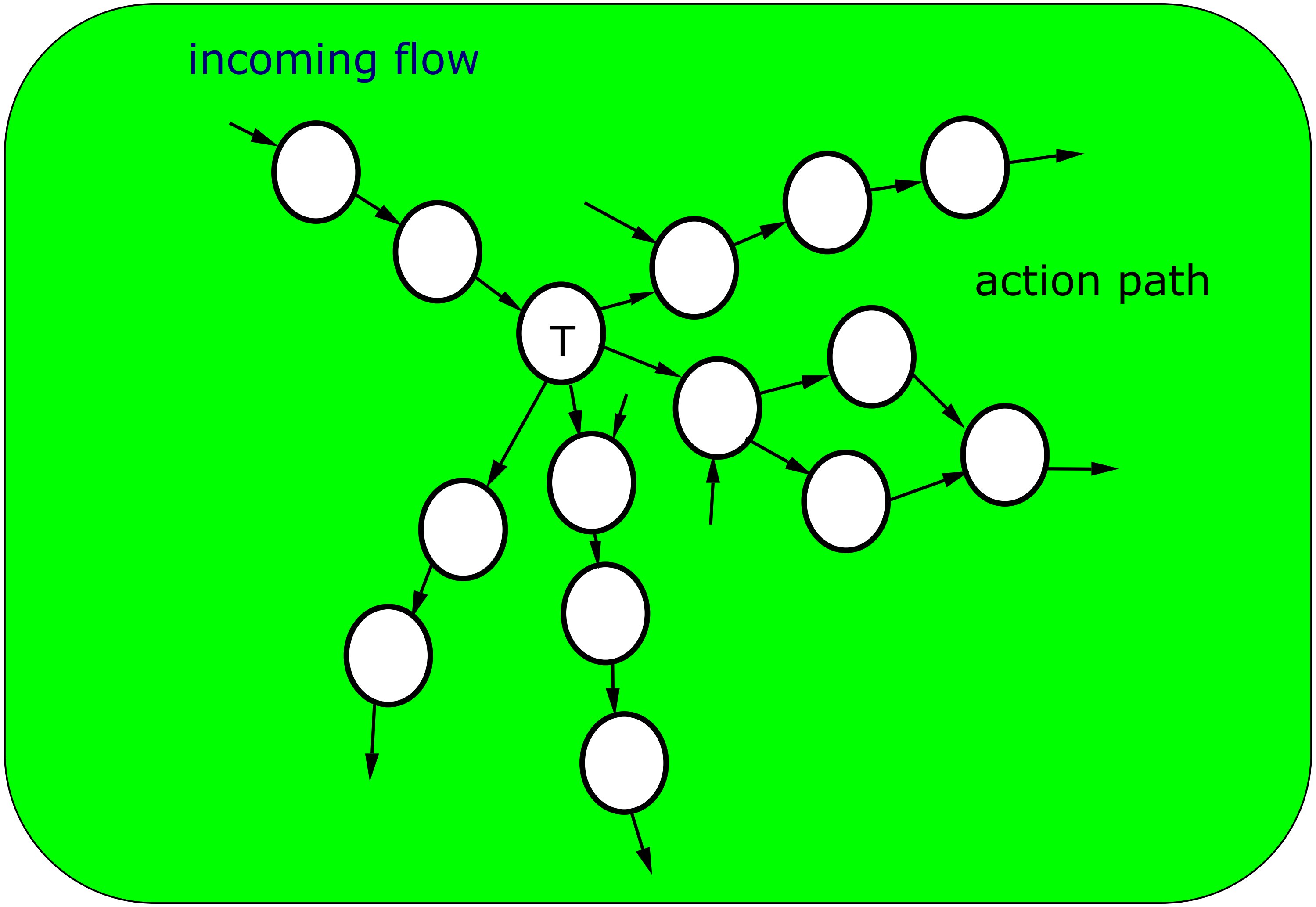


Transaction Mapping

- Review fundamental system model
- Review and refine data flow diagrams for the software
- Determine whether the DFD has transform or transaction characteristics

- Identify the transaction center and flow characteristics along each action path
- Map the DFD to a program structure amenable to transaction processing
- Factor and refine the transaction structure and the structure of each action path
- Refine the first iteration architecture using design heuristics for improved software quality

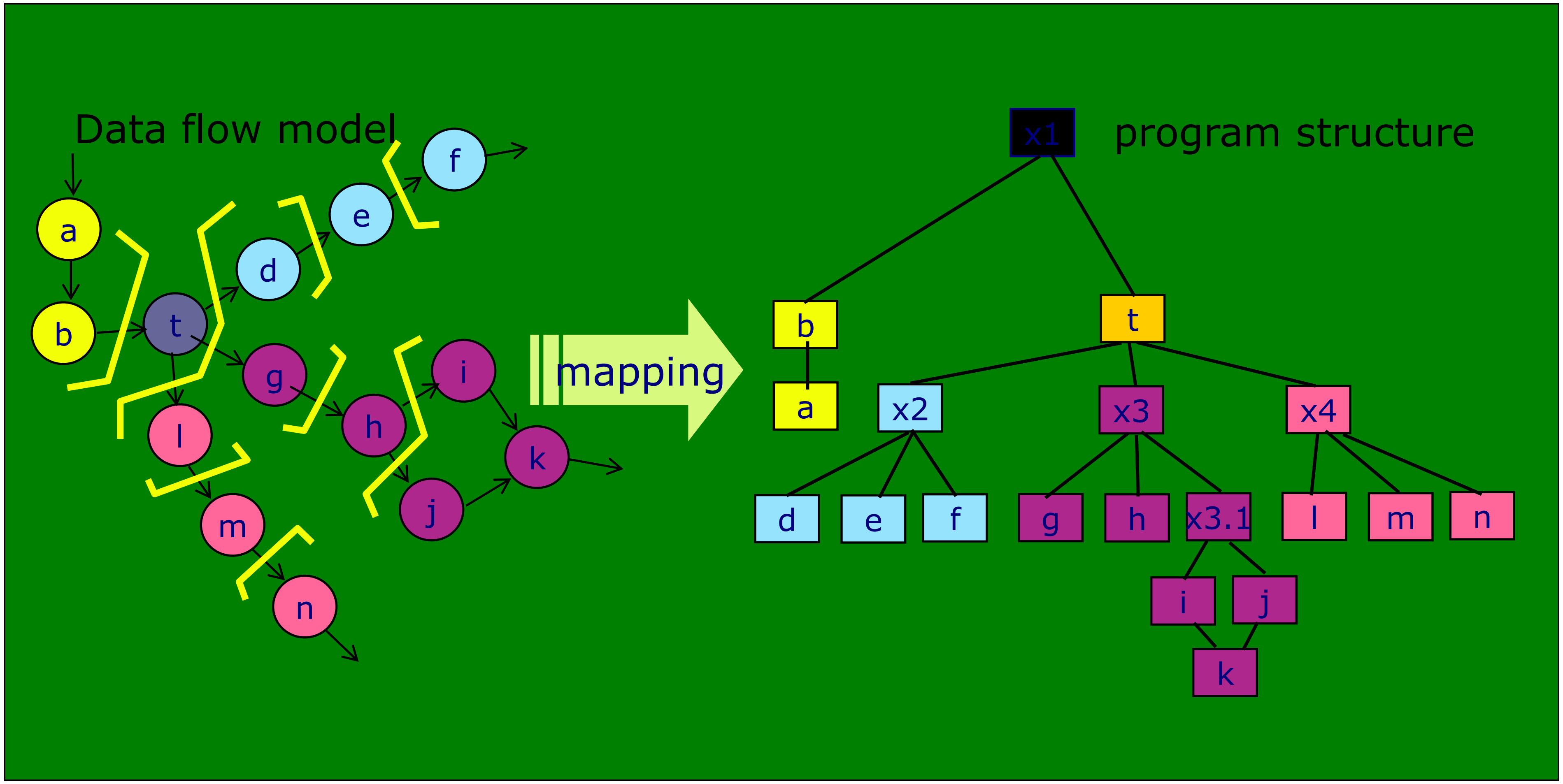
Transaction Flow



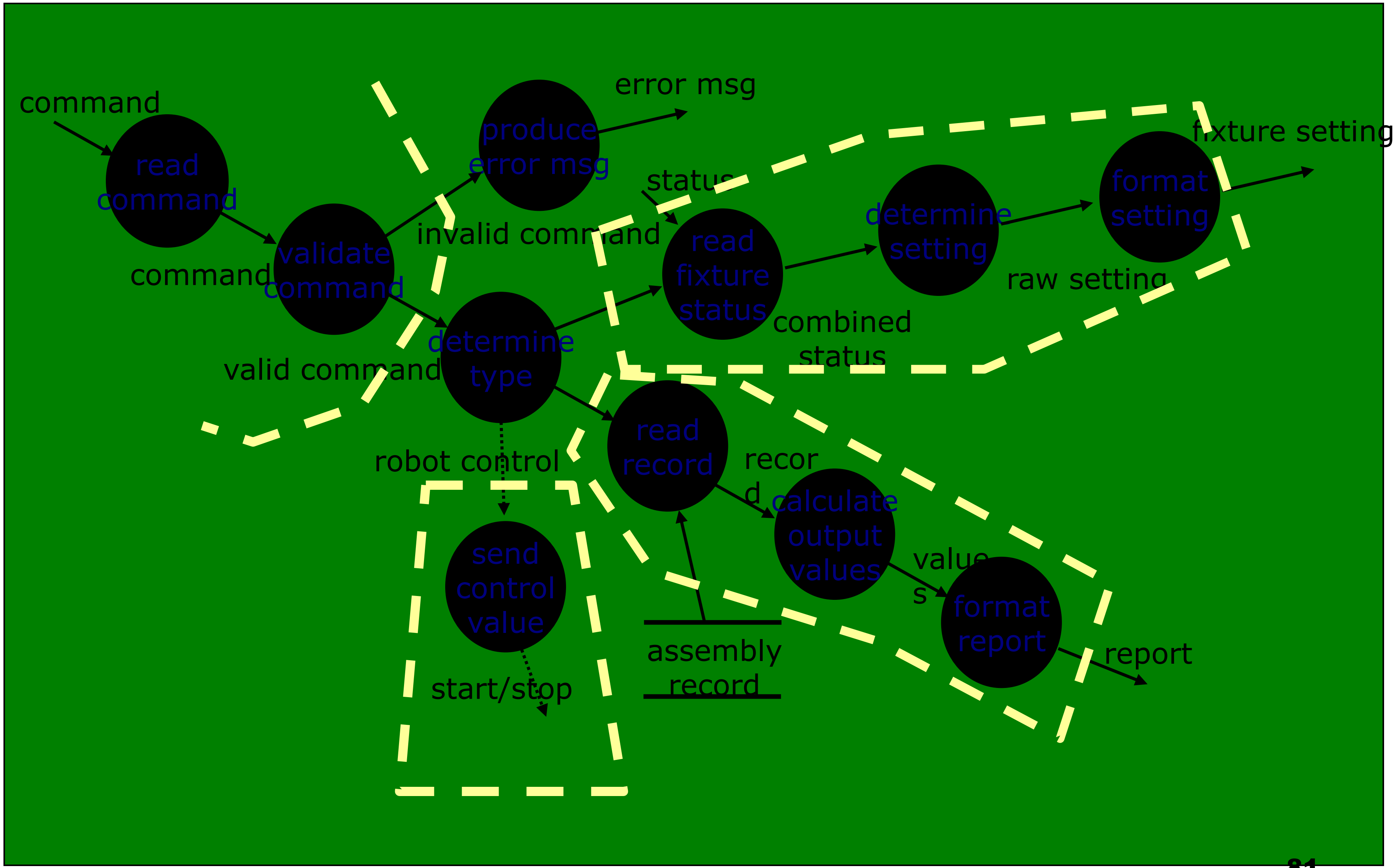
•Transaction Mapping Principles

- ❑ isolate the incoming flow path
- ❑ define each of the action paths by looking for the "spokes of the wheel"
- ❑ assess the flow on each action path
- ❑ define the dispatch and control structure
- ❑ map each action path flow individually

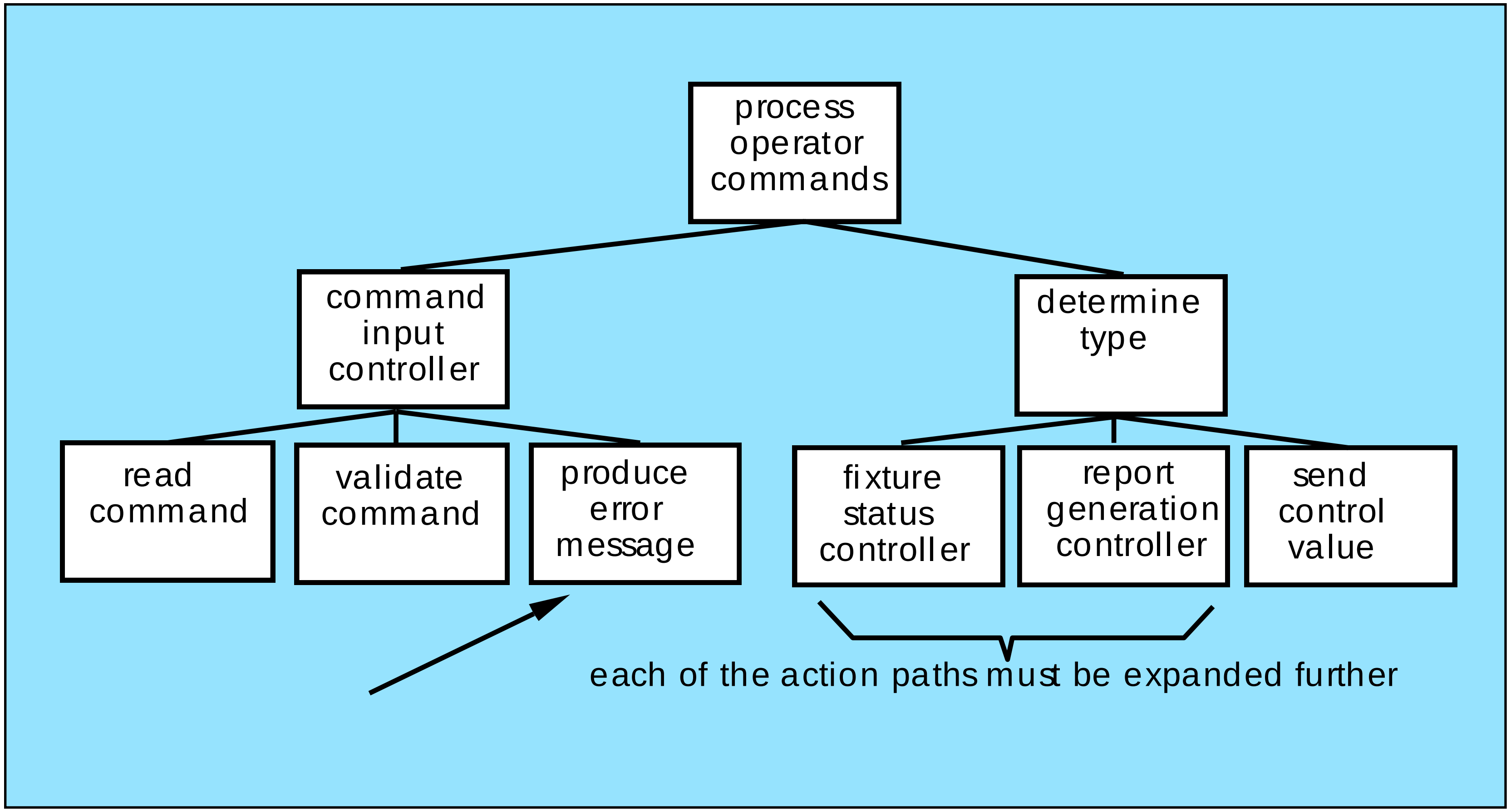
Transaction Mapping



Isolate Flow Paths



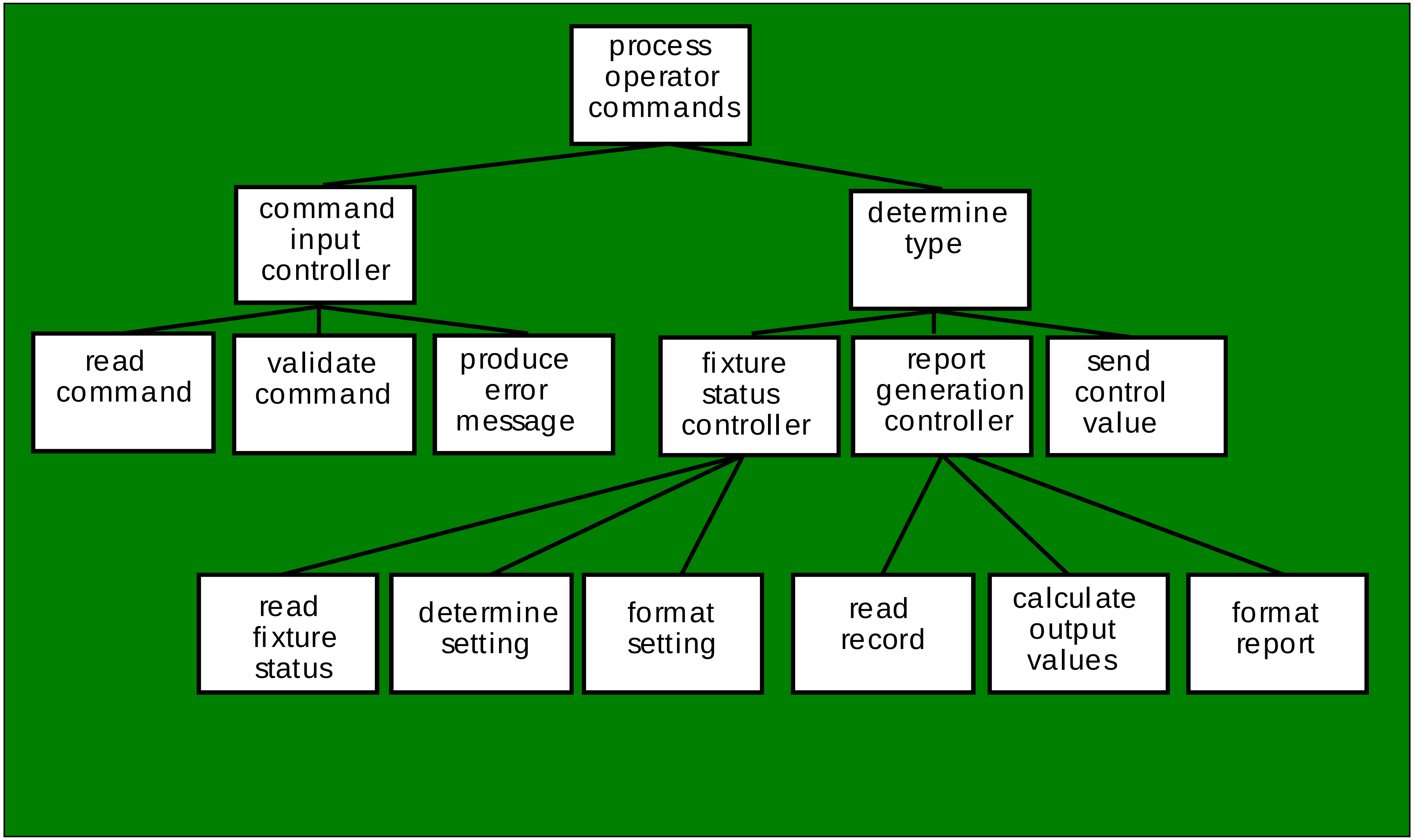
Map the Flow Model


process operator commands
command input controller
determine type
read command
validate command
produce error message
fixture status controller
report generation controller
send control value
each of the action paths must be expanded further

Refining Architectural Design

- Processing narrative developed for each module
- Interface description provided for each module
- Local and global data structures are defined
- Design restrictions/limitations noted
- Design reviews conducted
- Refinement considered if required and justified

Refining the Structure Chart



User Interface Design

Easy to learn?

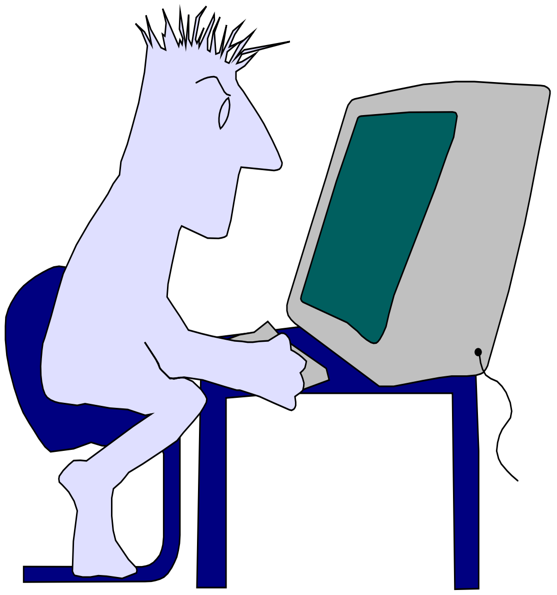
Easy to use?

Easy to understand?

Interface Design

<u>Typical Design Errors</u>

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 Load

- 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.

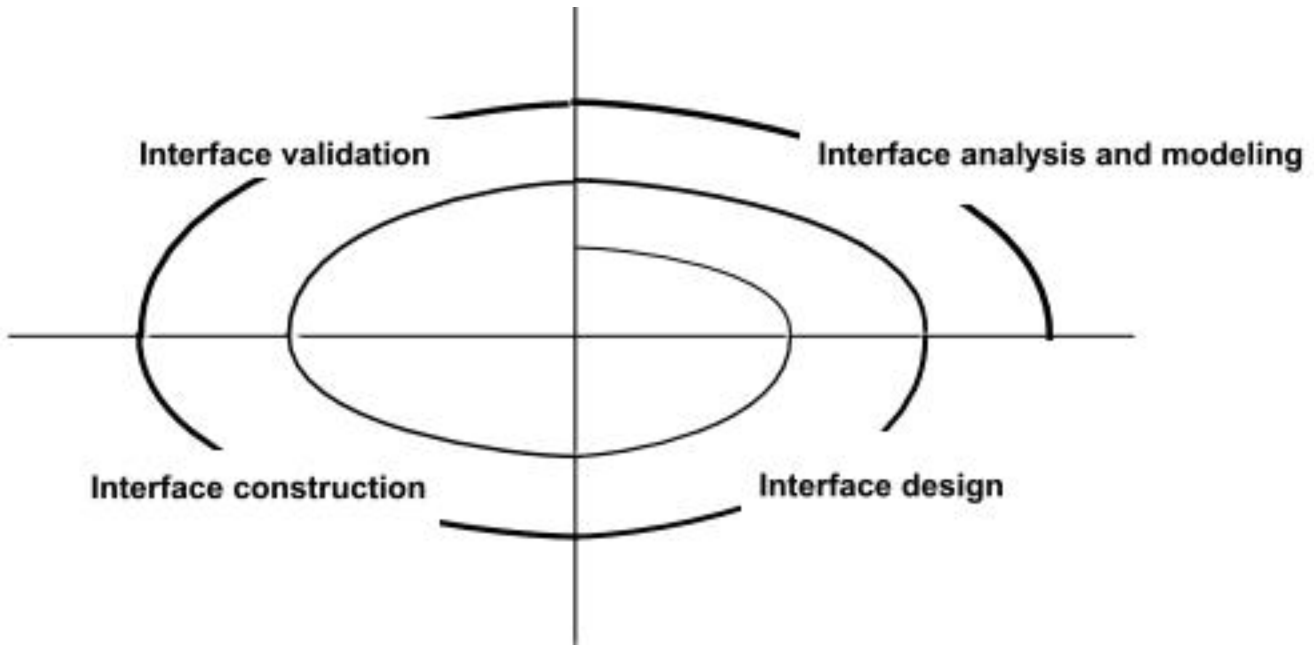
Make the Interface Consistent

- 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

- User model — a profile of all end users of the system
- Design model — a design realization of the user model
- Mental model (system perception) — the user's mental image of what the interface is
- Implementation model — the interface "look and feel" coupled with supporting information that describe interface syntax and semantics

User Interface Design Process



Interface Analysis

- Interface analysis means understanding
  - (1) the people (end-users) who will interact with the system through the interface;
  - (2) the tasks that end-users must perform to do their work,
  - (3) the content that is presented as part of the interface
  - (4) the environment in which these tasks will be conducted.

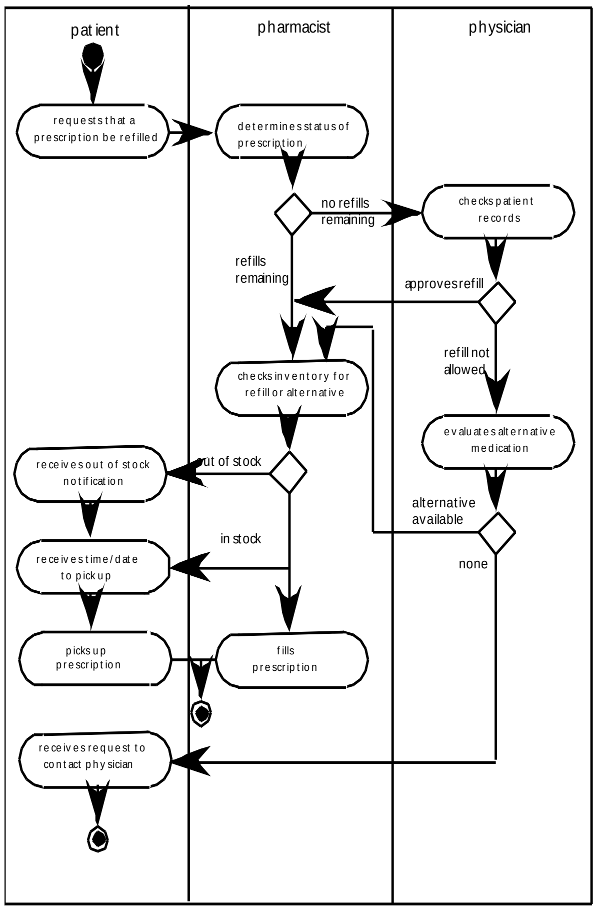
User Analysis

- Are users trained professionals, technician, clerical, or manufacturing workers?
- What level of formal education does the average user have?
- Are the users capable of learning from written materials or have they expressed a desire for classroom training?
- Are users expert typists or keyboard phobic?
- What is the age range of the user community?
- Will the users be represented predominately by one gender?
- How are users compensated for the work they perform?
- Do users work normal office hours or do they work until the job is done?
- Is the software to be an integral part of the work users do or will it be used only occasionally?
- What is the primary spoken language among users?
- What are the consequences if a user makes a mistake using the system?
- Are users experts in the subject matter that is addressed by the system?
- Do users want to know about the technology the sits behind the interface?

Task Analysis and Modeling

- Answers the following questions …
  - What work will the user perform in specific circumstances?
  - What tasks and subtasks will be performed as the user does the work?
  - What specific problem domain objects will the user manipulate as work is performed?
  - What is the sequence of work tasks—the workflow?
  - What is the hierarchy of tasks?
- Use-cases define basic interaction
- Task elaboration refines interactive tasks
- Object elaboration identifies interface objects (classes)
- Workflow analysis defines how a work process is completed when several people (and roles) are involved

Swimlane Diagram


patient
pharmacist
physician
requests that a prescription be refilled
determines status of prescription
no refills remaining
checks patient records
refills remaining
approves refill
refill not allowed
checks inventory for refill or alternative
evaluates alternative medication
receives out of stock notification
out of stock
alternative available
in stock
none
receives time/date to pick up
picks up prescription
fills prescription
receives request to contact physician

Analysis of Display Content

- Are different types of data assigned to consistent geographic locations on the screen (e.g., photos always appear in the upper right hand corner)?
- Can the user customize the screen location for content?
- Is proper on-screen identification assigned to all content?
- If a large report is to be presented, how should it be partitioned for ease of understanding?
- Will mechanisms be available for moving directly to summary information for large collections of data.
- Will graphical output be scaled to fit within the bounds of the display device that is used?
- How will color to be used to enhance understanding?
- How will error messages and warning be presented to the user?

Interface Design Steps

- Using information developed during interface analysis, define interface objects and actions (operations).
- Define events (user actions) that will cause the state of the user interface to change. Model this behavior.
- Depict each interface state as it will actually look to the end-user.
- Indicate how the user interprets the state of the system from information provided through the interface.

Design Issues

- Response time
- Help facilities
- Error handling
- Menu and command labeling
- Application accessibility
- Internationalization

Component-Level Design

What is Comp. Level Design?

- A complete set of software components is defined during architectural design
- But the internal data structures and processing details of each component are not represented at a level of abstraction that is close to code
- Component-level design defines the data structures, algorithms, interface characteristics, and communication mechanisms allocated to each component

What is a component?

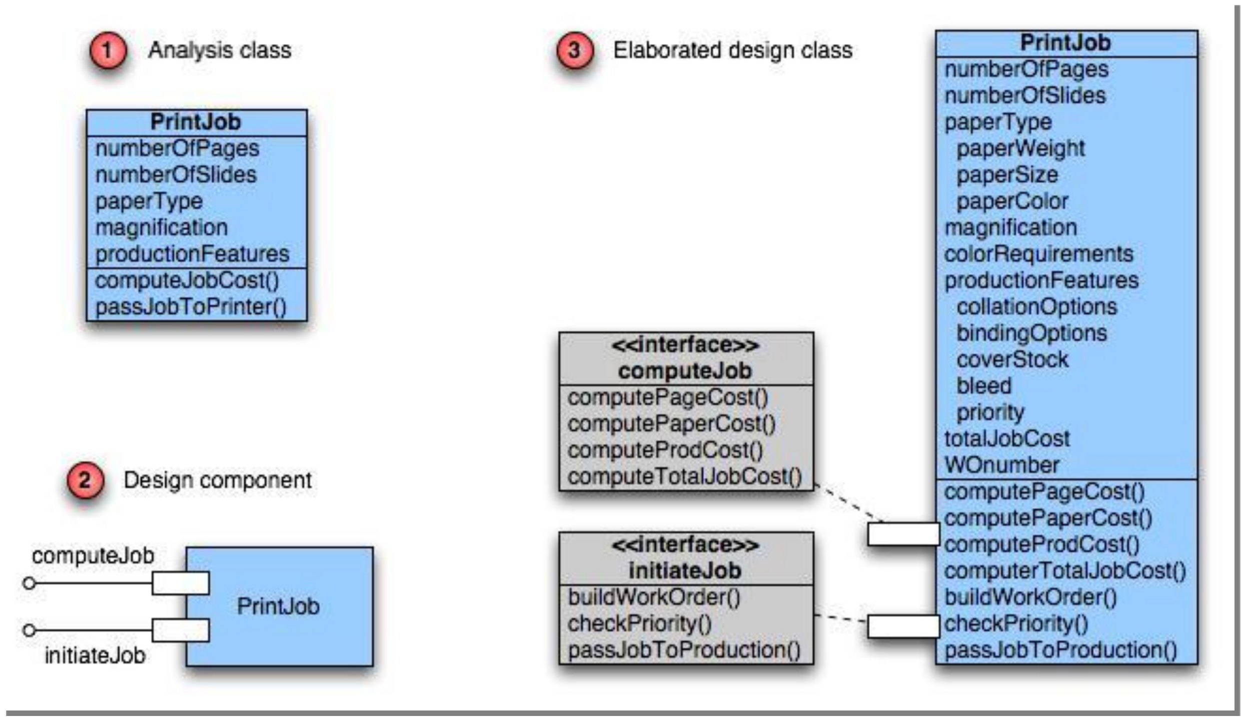
- "A modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces."

— *OMG UML Specification*

Component Views

- OO View – A component is a set of collaborating classes.
- Conventional View – A component is a functional element of a program that incorporates processing logic, the internal data structures required to implement the processing logic, and an interface that enables the component to be invoked and data to be passed to it.

Class Elaboration



Design Principles

- Design by Contract
- Open-Closed Principle
- Subtype Substitution
- Depend on Abstractions
- Interface Segregation

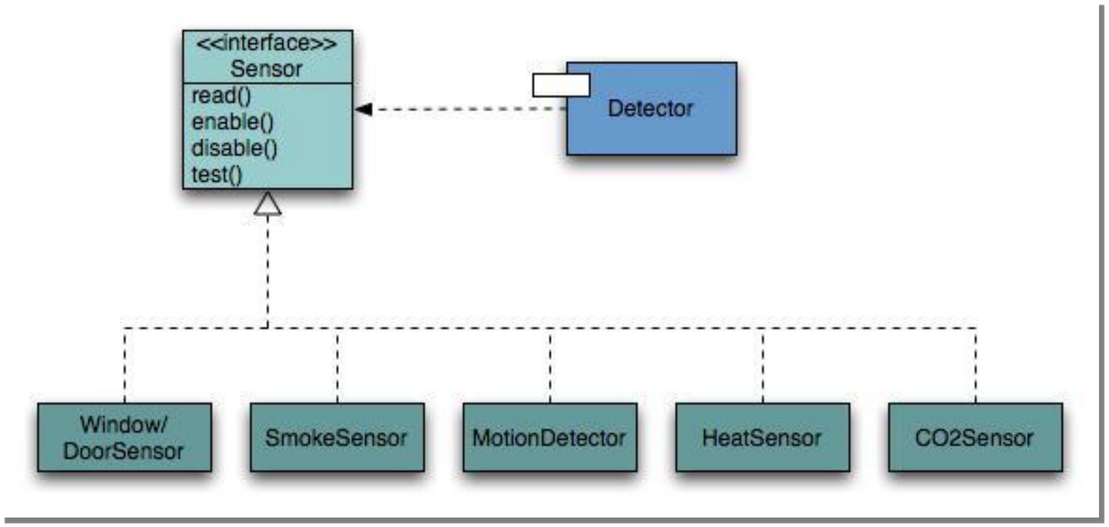
Design by Contract

- The relationship between a class and its clients can be viewed as a formal agreement, expressing each party's rights and obligations.
- Consider the following list operation:

- public Item remove(int index)
  - requires the specified index is in range ($0 \leq$ index < size( ))
  - ensures the element at the specified position in this list is removed, subsequent elements are shifted to the left (1 is subtracted from their indices), and the element that was removed is returned

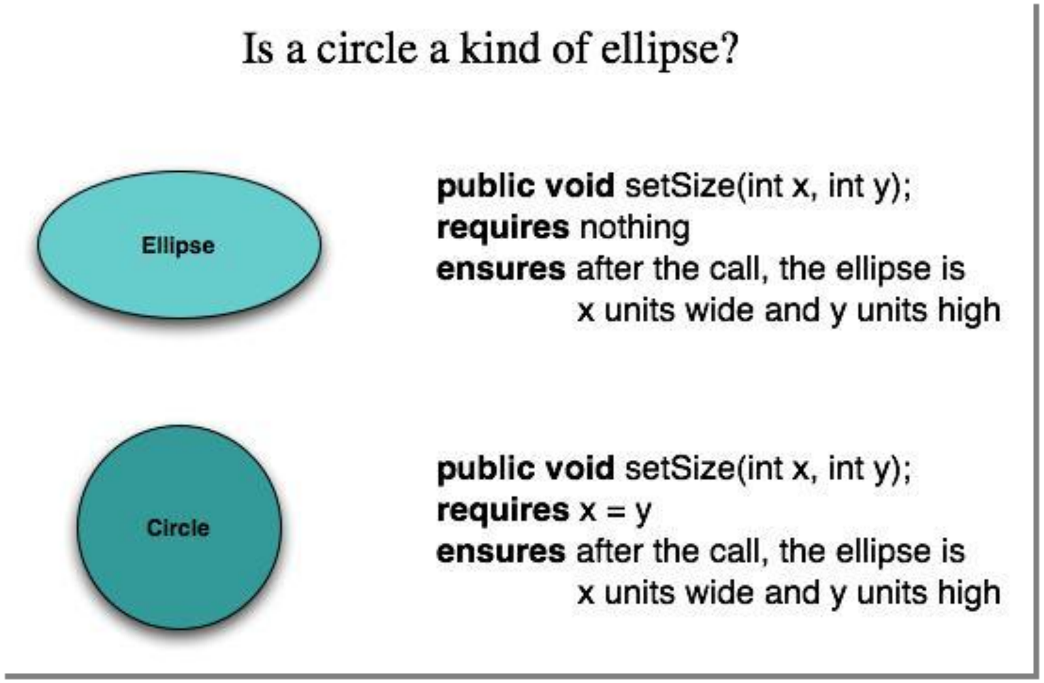
Open-Closed Principle

- A module should be open for extension but closed for modification.



Substitutability

- Subclasses should be substitutable for base classes

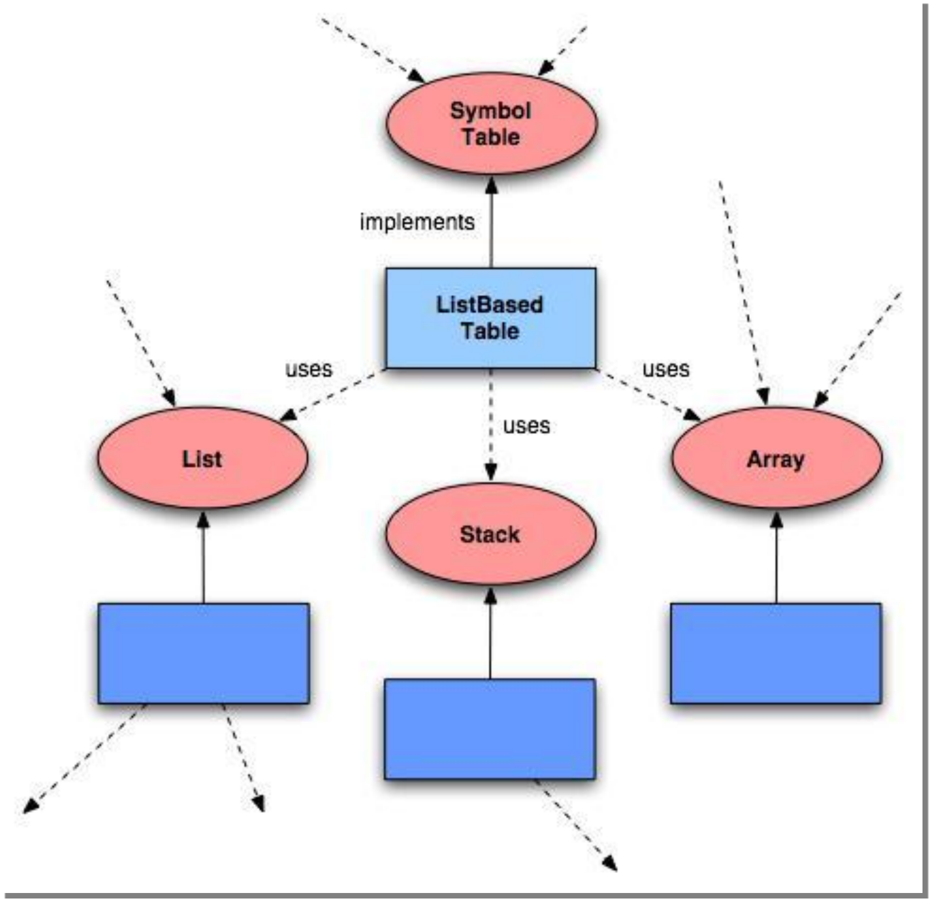


public void setSize(int x, int y);
requires nothing
ensures after the call, the ellipse is
x units wide and y units high

public void setSize(int x, int y);
requires x = y
ensures after the call, the ellipse is
x units wide and y units high

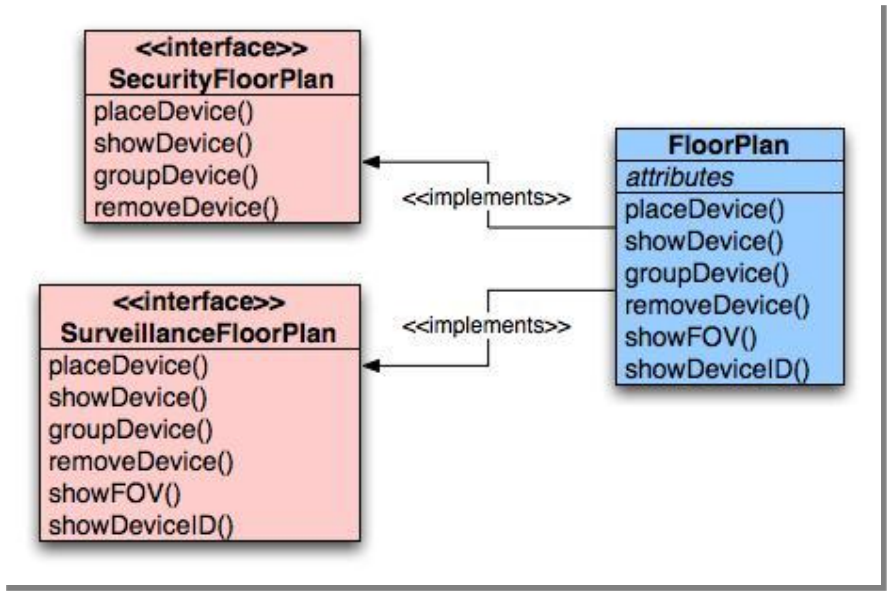
Dependency Inversion

- Depend on abstractions. Do not depend on concretions.



Interface Segregation

- Many client-specific interfaces are better than one general purpose interface.

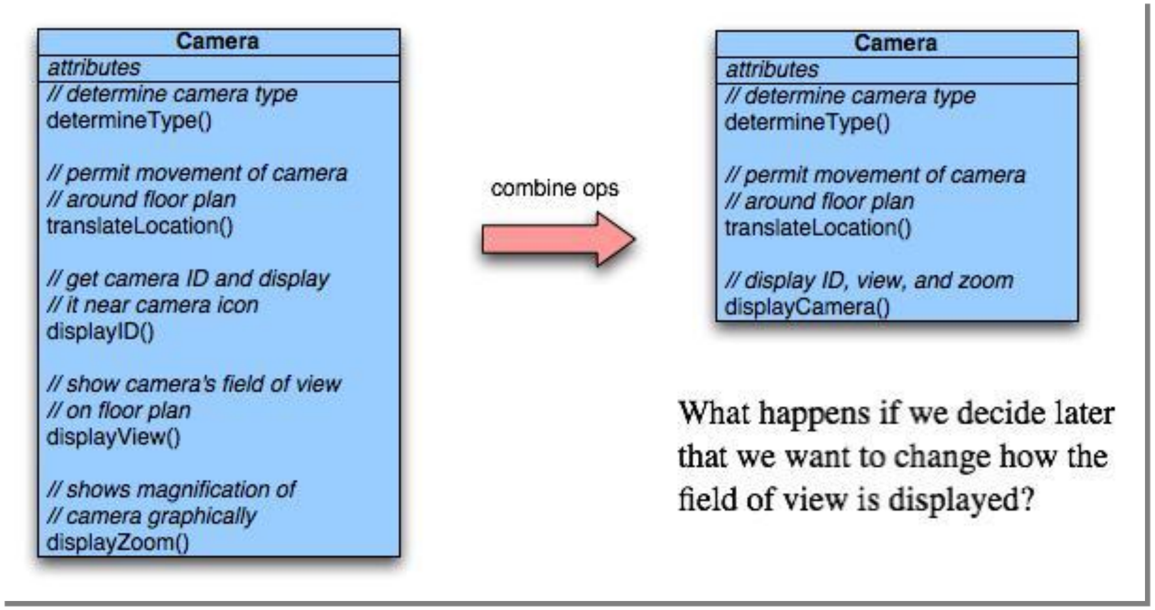


Cohesion

- The “single-mindedness” of a module
- cohesion implies that a single component or class encapsulates only attributes and operations that are closely related to one another and to the class or component itself.
- Examples of cohesion
  - Functional
  - Layer
  - Communicational

Functional Cohesion

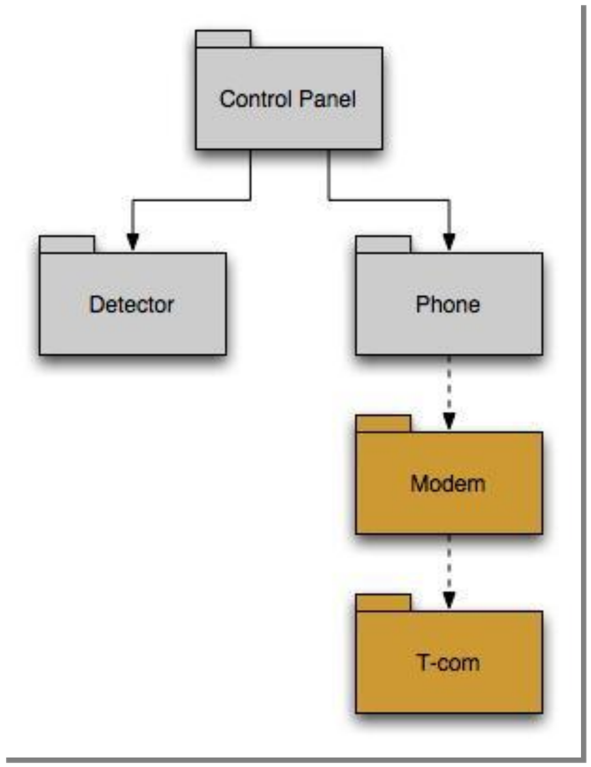
- Typically applies to operations. Occurs when a module performs one and only one computation and then returns a result.



What happens if we decide later that we want to change how the field of view is displayed?

Layer Cohesion

- Applies to packages, components, and classes. Occurs when a higher layer can access a lower layer, but lower layers do not access higher layers.



Communicational Cohesion

- All operations that access the same data are defined within one class.
- In general, such classes focus solely on the data in question, accessing and storing it.
- Example: A StudentRecord class that adds, removes, updates, and accesses various fields of a student record for client components.

Coupling

- A qualitative measure of the degree to which classes or components are connected to each other.
- Avoid
  - Content coupling
- Use caution
  - Common coupling
- Be aware
  - Routine call coupling
  - Type use coupling
  - Inclusion or import coupling

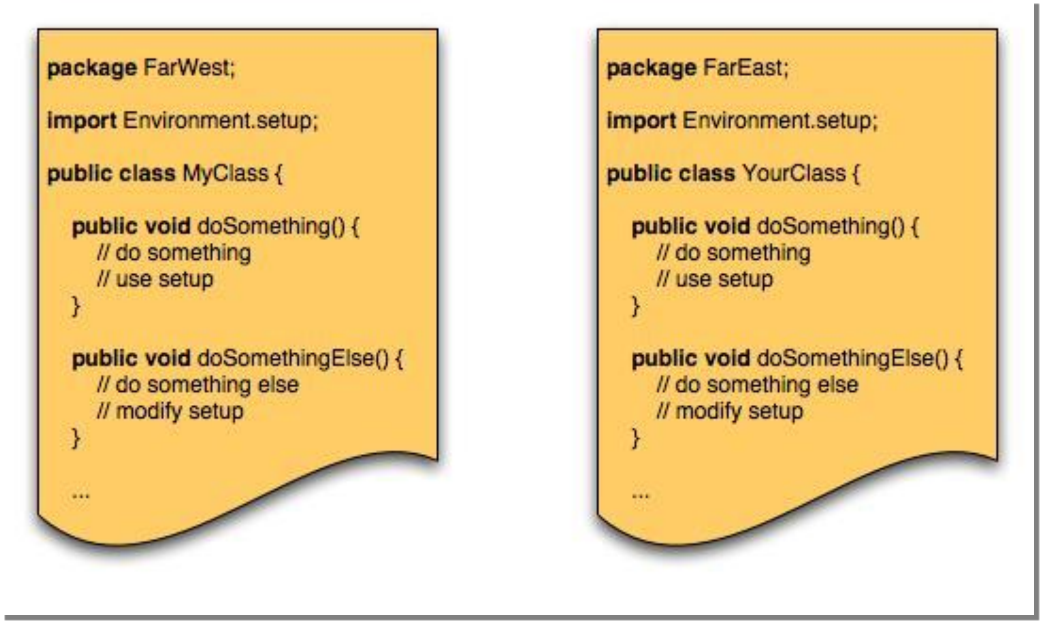
Content Coupling

- Occurs when one component "surreptitiously modifies data that is internal to another component"
- Violates information hiding
- What's wrong here? →

```
public class StudentRecord {

    private String name;
    private int[ ] quizScores;

    public String getName() {
        return name;
    }
    public int getQuizScore(int n) {
        return quizScores[n];
    }
    public int[ ] getAllQuizScores() {
        return quizScores;
    }

    ....
```

Common Coupling

- Occurs when a number of components all make use of a global variable.

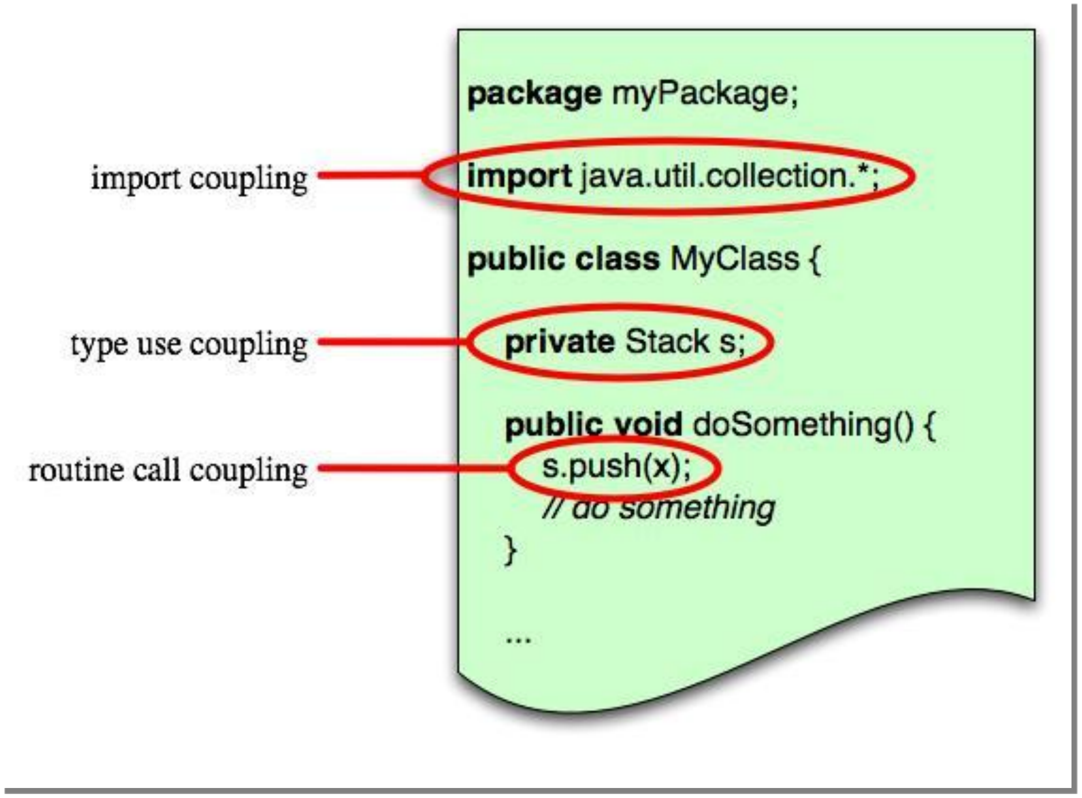


```
package FarWest;

import Environment.setup;

public class MyClass {

    public void doSomething() {
        // do something
        // use setup
    }

    public void doSomethingElse() {
        // do something else
        // modify setup
    }

    ...
```

```
package FarEast;

import Environment.setup;

public class YourClass {

    public void doSomething() {
        // do something
        // use setup
    }

    public void doSomethingElse() {
        // do something else
        // modify setup
    }

    ...
```

Routine Coupling

- Certain types of coupling occur routinely in object-oriented programming.



```
package myPackage;

import java.util.collection.*;

public class MyClass {

  private Stack s;

  public void doSomething() {
    s.push(x);
    // do something
  }

  ...
```

Component-Level Design

1. Identify design classes in problem domain
2. Identify infrastructure design classes
3. Elaborate design classes
4. Describe persistent data sources
5. Elaborate behavioral representations
6. Elaborate deployment diagrams
7. Refactor design and consider alternatives

Designing Traditional components

- Graphical Design Notation
- Tabular Design Notation
- Program Design Language