Career Mentorship Colloquium: Working at RDI

Brian Walch, Jason Mancuso
Resource Data, Inc.

Date: November 10, 2011
Time: 2:00 p.m.
Location: 260 Deschutes

Abstract

Brian Walch and Jason Mancuso from Resource Data, Inc. will talk about working at RDI. RDI is an information technology company specializing in GIS and custom software and system design and implementation. They will also discuss current and future opportunities for employment with RDI.

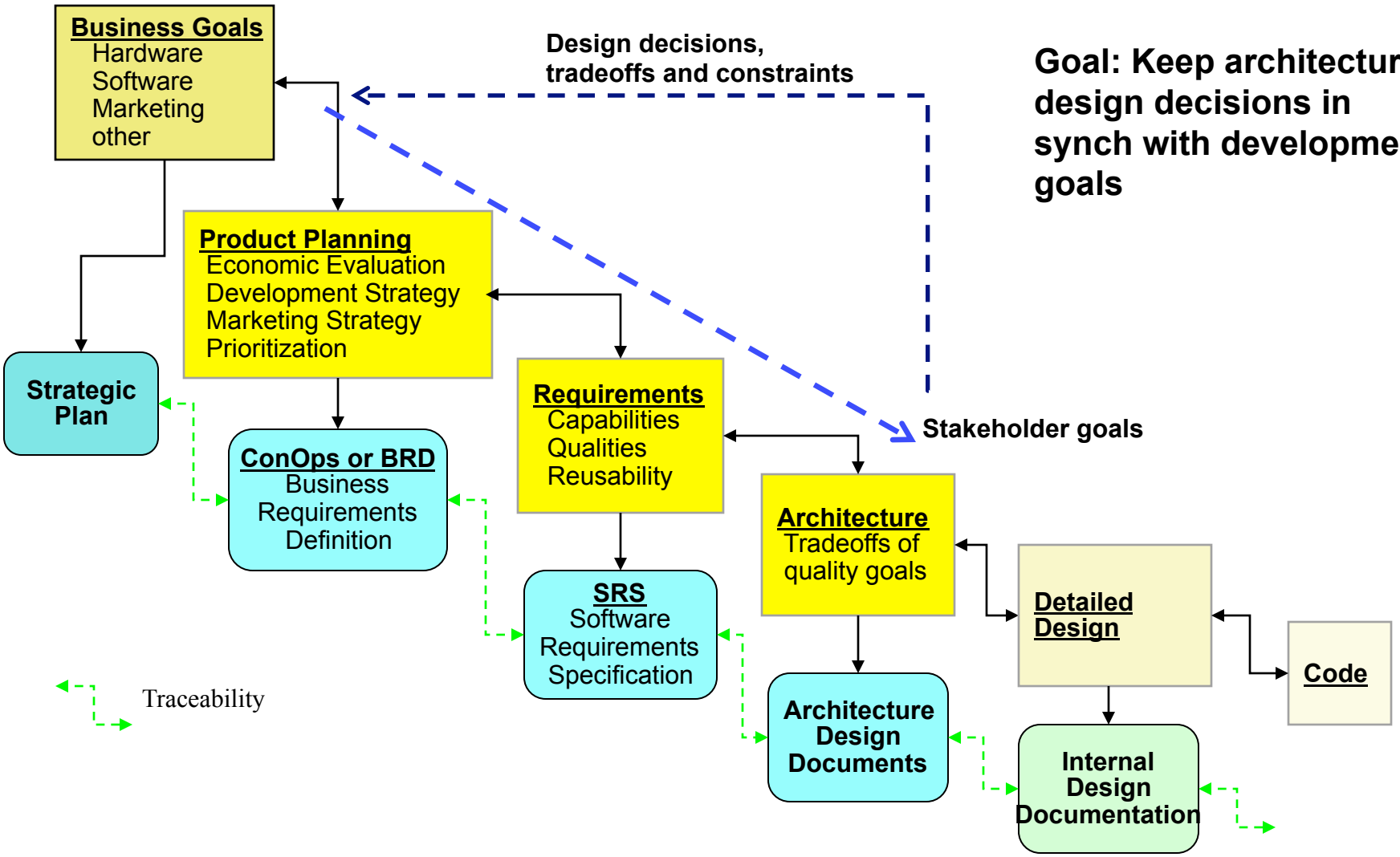
Achieving System Qualities Through Software Architecture

The meaning of “design”

The module structure

Design principles

Product Development Cycle and Architecture



SW Engineering of Software Architecture

- What are we trying to gain/maintain control of in the Architectural Design phase?
  - Profoundly effect system and business qualities
  - Requires making tradeoffs
- Control implies achieving system qualities by choice not chance
  - Understanding what the tradeoffs are
  - Understanding the consequences of each choice
  - Making appropriate choices at appropriate times

Implications for the Development Process

Implies need to address architectural concerns in the development process:

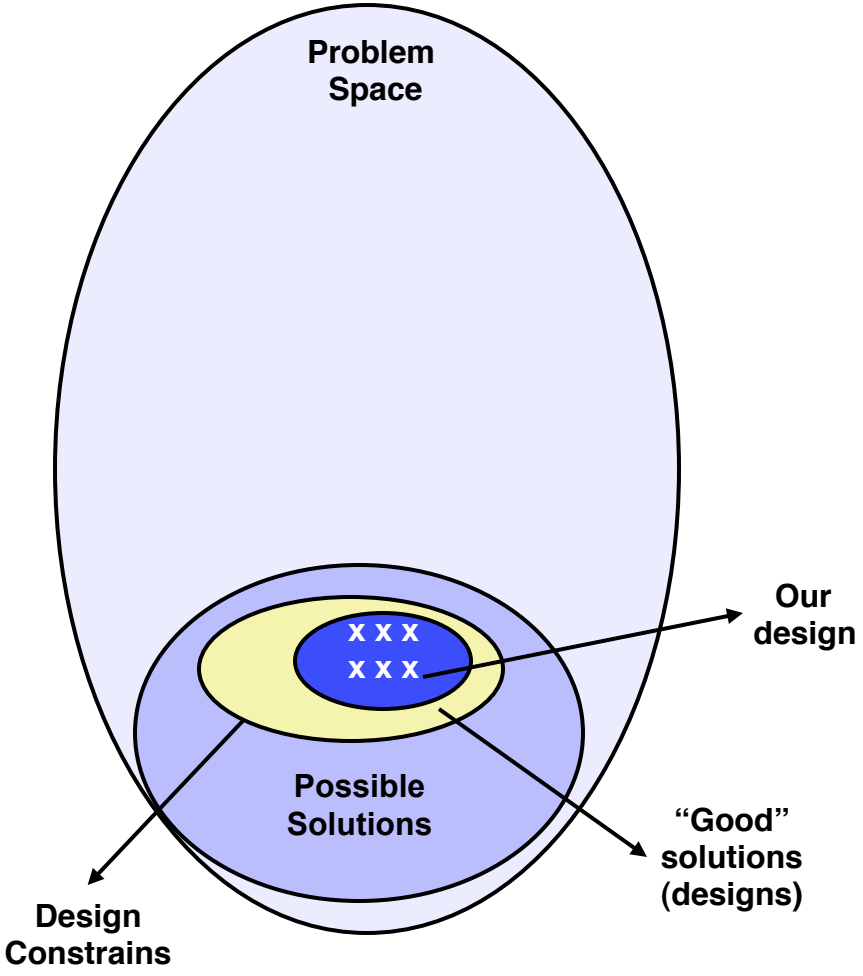
- Understanding the "business case" for the system
- Understanding the quality requirements
- **Designing the architecture**
- Representing and communicating the architecture
- Analyzing or evaluating the architecture
- Implementing the system based on the architecture
- Ensuring the implementation conforms to the architecture

What is “design?”

Meaning of “Design”

- What does it mean to say that we are going to “design the software?”
- What is the basis for making a design decision?
- How do we know when we are done?
- If we did a good job? What makes a good design?

The Design Space



- A Design: is (a representation of) a solution to a problem
  - Represents a set of choices
    - Typically very large set of possible choices
    - Must navigate through possibilities
    - Invariably requires tradeoffs
  - Possible choices are limited by *assumptions and constraints*
    - e.g., must be ISO 2000 compliant, legacy compatible, etc.
  - Some designs are better than others (notion of *good design*)

Design Means…

- Design Goals: the purpose of design is to solve some problem in a context of assumptions and constraints
  - Assumptions: what must be true of the design
  - Constraints: what should not be true
  - **These define the *design goals***
- Process: design proceeds through a sequence of decisions
  - A *good* decision brings us closer to the design goals
  - An idealized design process systematically makes good decisions
  - Any real design process is chaotic
- Good Design: *by definition* a good design is one that satisfies the design goal

Architectural Design Elements

- Design goals
  - What are we trying to accomplish in the decomposition?
- Relevant Structure
  - How to we capture and communicate design decisions?
  - What are the components, relations, interfaces?
- Decomposition principles
  - How do we distinguish good design decisions?
  - What decomposition (design) principles support the objectives?
- Evaluation criteria
  - How do I tell a good design from a bad one?

Examples of Key Architectural Structures

- Module Structure
  - Decomposition of the system into work assignments or information hiding modules
  - Most influential design time structure
    - Modifiability, work assignments, maintainability, reusability, understandability, etc.
- Uses Structure
  - Determine which modules may use one another's services
  - Determines subsetability, ease of integration

Designing the Module Structure

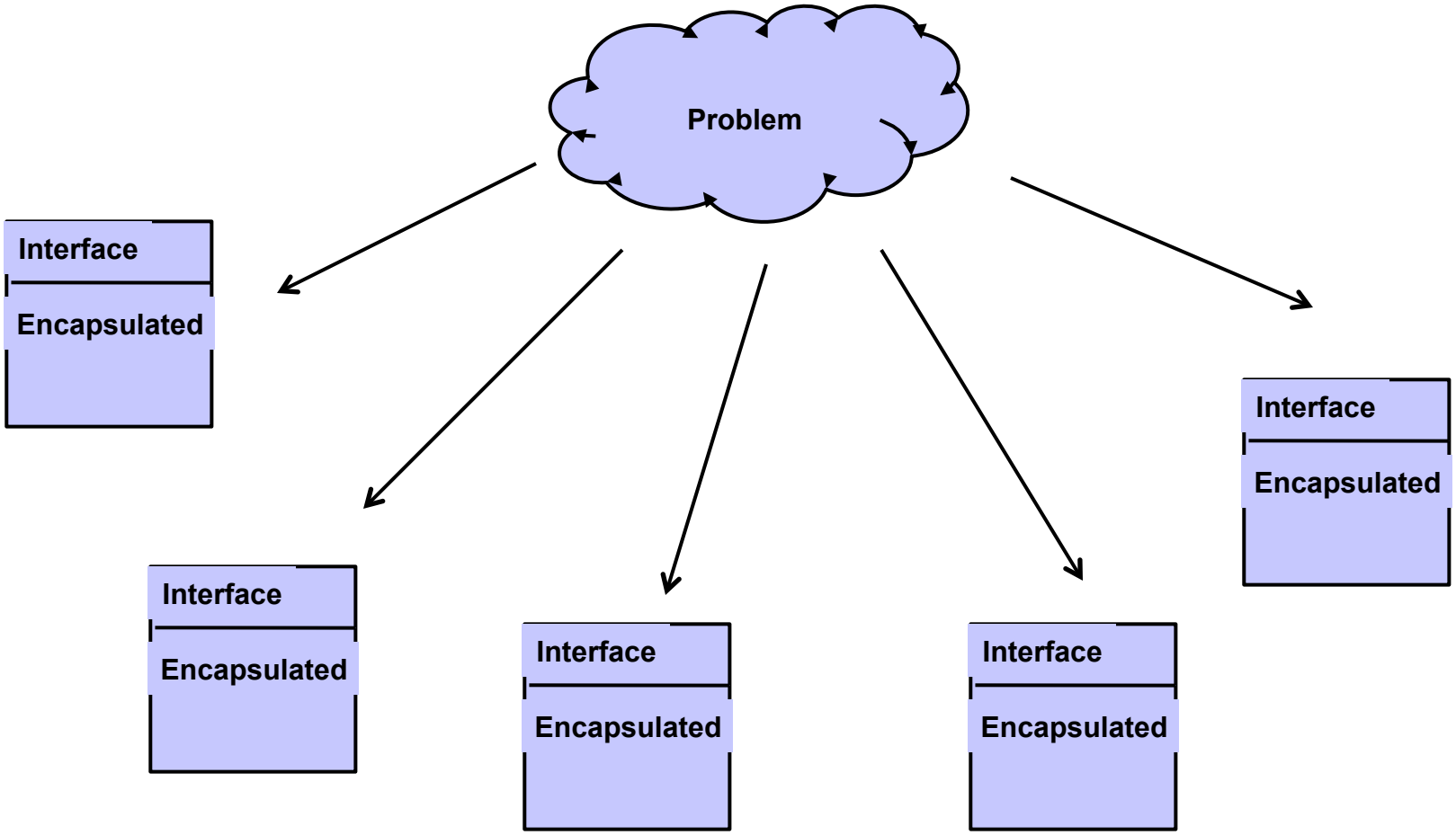
Modularization

- For large, complex software, must divide the development into work assignments (WBS). Each work assignment is called a “module.”
- Properties of a “good” module structure
  - Parts can be designed independently
  - Parts can be tested independently
  - Parts can be changed independently
  - Integration goes smoothly

Expected Control Improvements

- Reduces complexity, improves manageability
- Coding
  - Can write modules with little knowledge of other modules
  - Replace modules without reassembling the whole system
- Managerial
  - Allows concurrent development
  - Avoids “Mythical Man Month” effect (“adding people to a late software project makes it later”)
- Flexibility/Maintainability
  - Anticipated changes affect only a small number of modules (usually one)
  - Can calculate the impact and cost of change
- Review/communicate
  - Can understand or review the system one module at a time

Notional Modules

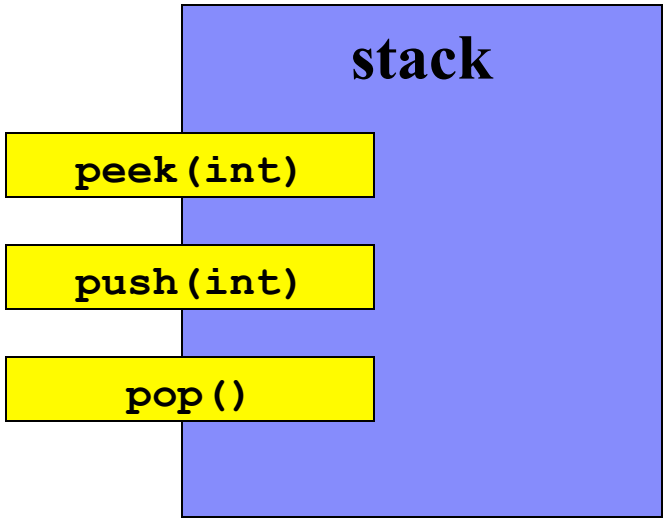


What is a module?

- Concept due to David Parnas (conceptual basis for objects)
- A module is characterized by two things:
  - Its interface: services that the module provides to other parts of the systems
  - Its secrets: what the module hides (encapsulates). Design/ implementation decisions that other parts of the system should not depend on
- Modules are abstract, design-time entities
  - Modules are “black boxes” – specifies the visible properties but not the implementation
  - May or may not directly correspond to programming components like classes/objects
    - E.g., one module may be implemented by several objects

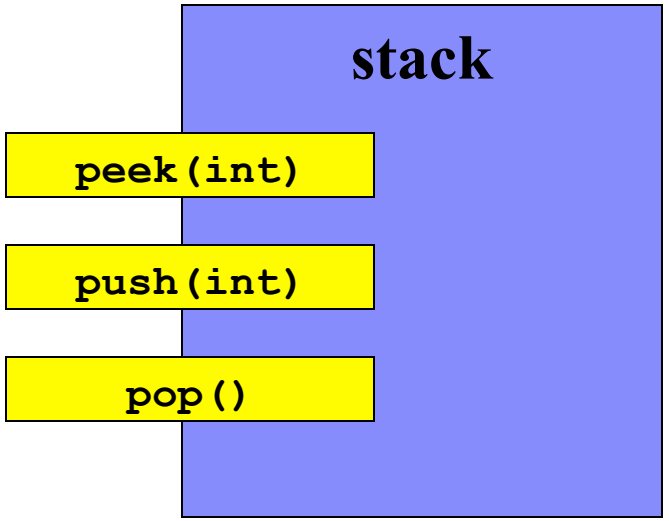
A Simple Module

- A simple integer stack
  - *push*: push integer on stack top
  - *pop*: remove top element
  - *peek*: get value of top element
- What information is on the interface?
- What are the secrets?
- What information is missing?
- Why is this an abstraction?



A Simple Module

- A simple integer stack
- The *interface* specifies what a programmer needs to know to use the stack correctly, e.g.
  - *push*: push integer on stack top
  - *pop*: remove top element
  - *peek*: get value of top element
- The *secrets* (encapsulated) any details that might change from one implementation to another
  - Data structures, algorithms
  - Details of class/object structure
- A module spec is *abstract*: describes the services provided but allows many possible implementations
- Note: a real spec needs much more than this (discuss later)



Why these properties?

Module Implementer

- The specification tells me exactly what capabilities my module must provide to users
- I am free to implement it any way I want to
- I am free to change the implementation if needed as long as I don't change the interface

Module User

- The specification tells me how to use the module's services correctly
- I do not need to know anything about the implementation details to write my code
- If the implementation changes, my code stays the same

***Key idea*: the abstract interface specification defines a contract between a module's developer and its users that allows each to proceed independently**

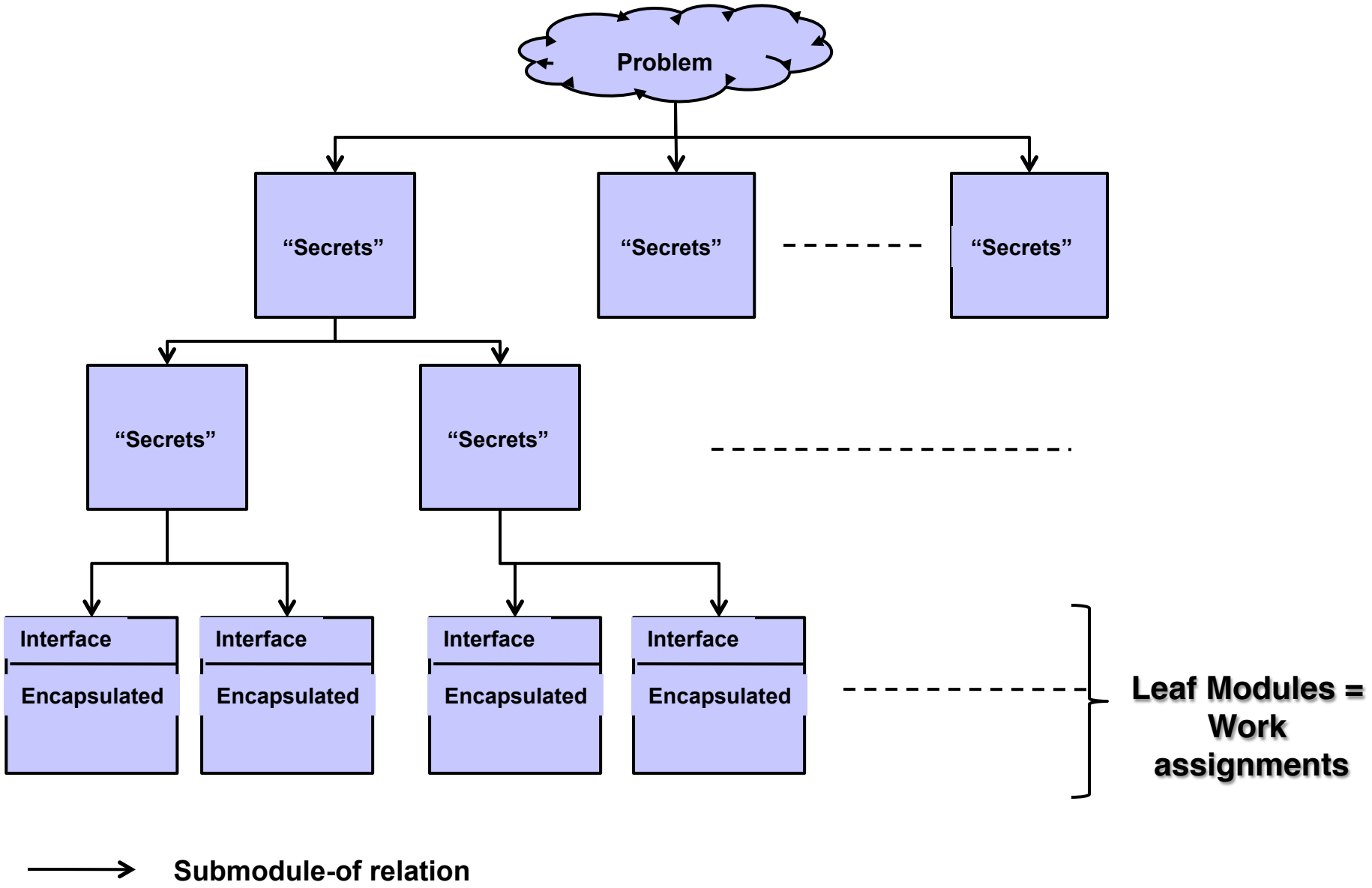
Is a module a class/object?

- The programming language concepts of classes and objects are based on Parnas' concept of modules
- To separate design-time concerns from coding issues, however, *they are not the same thing*
  - A module must be a work assignment for a team at design time, it should not determine the run-time structure of the code
  - The implementer must be free to implement with a different class structure as long as the interface capabilities are provided
  - The implementer must be free to make changes as long as the interface does not change
- In simple cases, we will often implement each module as a class/object

Module Hierarchy

- For large systems, the set of modules need to be organized such that
  - We can check that all of the functional requirements have been allocated to some module of the system
  - Developers can easily find the module that provides any given capability
  - When a change is required, it is easy to determine which modules must be changed
- The module hierarchy defined by the *submodule-of* relation provides this architectural view

Module Hierarchy



Decomposition Approach

Design Goals

- Recast as module structure design goals
- Divide software into set of work assignments with the following properties:
  - *Easy to Understand:* Each module's structure should be simple enough that it can be understood fully.
  - *Easy to Change (mutability):* It should be possible to change the implementation of one module without knowledge of the implementation or affecting the behavior of other modules.
  - *Proportion:* Effort of making a change should be in (reasonably) direct proportion to the likelihood of that change being necessary.
  - *Independence:* It should be possible to make a major change as a set of independent changes to individual modules

Modular Structure

- Comprises components, relations, and interfaces
- Components
  - Called modules
  - Leaf modules are work assignments
  - Non-leaf modules are the union of their submodules
- Relations (connectors)
  - submodule-of => implements-secrets-of
  - The union of all submodules of a non-terminal module must implement all of the parent module's secrets
  - Constrained to be acyclic tree (hierarchy)
- Interfaces (externally visible component behavior)
  - Defined in terms of access procedures (services or method)
  - Only external (exported) access to internal state

Decomposition Strategies Differ

- How do we develop this structure so that *we know* the leaf modules make independent work assignments?
- Many ways to decompose hierarchically
  - Functional: each module is a function
  - Steps in processing: each module is a step in a chain of processing
  - Data: data transforming components
  - Client/server
  - Use-case driven development
- But, these result in different kinds of dependencies (strong coupling)

Submodule-of Relation

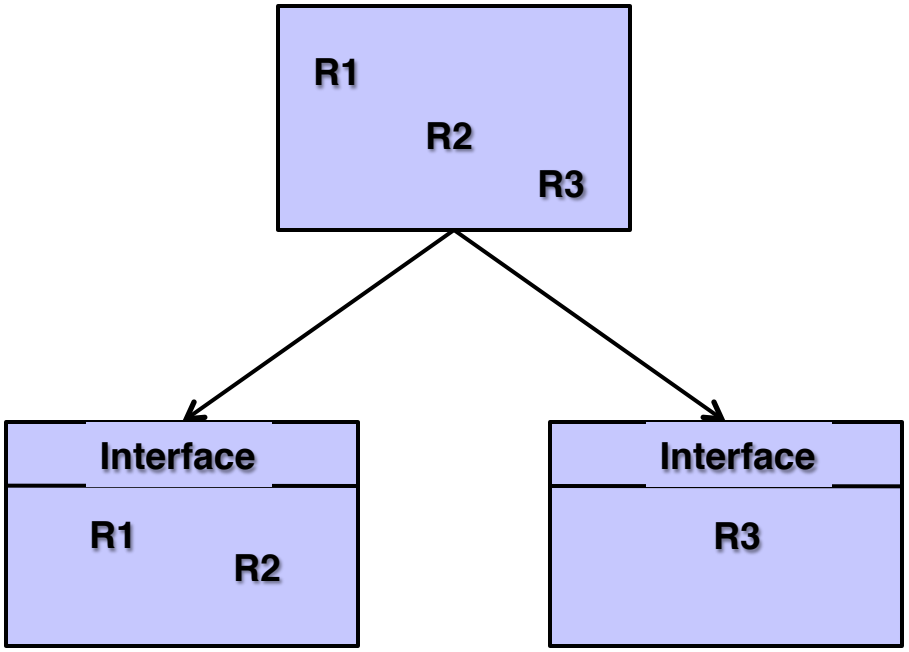
- To define the structure, need the *relation* and the *rule* for constructing the relation
- Relation: sub-module-of
- Rules
  - If a module consists of parts that can change independently, then decompose it into submodules
  - Don't stop until each module contains only things likely to change together
  - Anything that other modules should not depend on become secrets of the module (e.g., implementation details)
  - If the module has an interface, only things not likely to change can be part of the interface

Applied Information Hiding

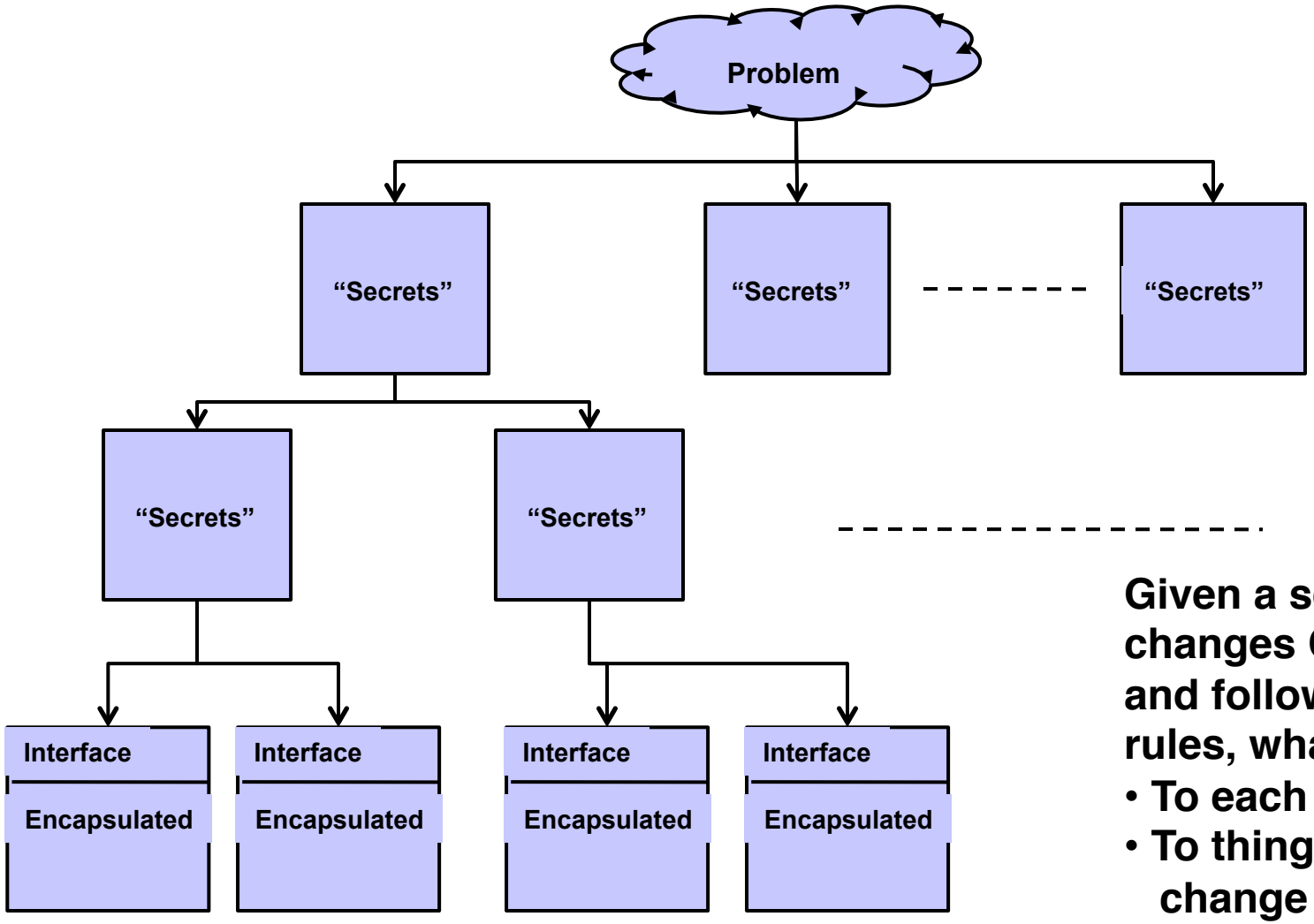
- The rule we just described is calls *the information hiding principle*
- Information hiding (or encapsulation): Design principle of limiting dependencies between components by hiding information other components should not depend on
- An information hiding decomposition is one following the design principles that:
  - System details that are likely to change independently are encapsulated in different modules
  - The interface of a module reveals only those aspects considered unlikely to change

Effects of Changes

- Consider what happens to communication among teams
- Suppose we have groups of requirements R1 – R3:
  - R1 and R3 are related and likely to change together
  - R2 is likely to change independently
- Suppose we put R1 and R2 in the same module and assign to different teams
  - What happens when R1 changes?
  - R2?
- Suppose R1 and R3 are put in the same module?



Module Hierarchy



Given a set of likely changes C1, C2, … Cn and following these rules, what happens:

- **To each change?**
- **To things that change together?**
- **Change separately?**

Submodule-of relation

Evaluation Criteria

- Evaluation criteria follow from goals of the model: should be able to answer “yes” to the following review questions?
- Completeness
  - Is every aspect of the system the responsibility of one module?
  - Do the submodules of each module partition its secrets?
- Ease change
  - Is each likely change hidden by some module?
  - Are only aspects of the system that are very unlikely to change embedded in the module structure?
  - For each leaf module, are the module’s secrets revealed by it’s access programs?
- Usability
  - For any given change, can the appropriate module be found using the module guide

Module Decomposition

- Approach: divide the system into submodules according to the kinds of design decisions they encapsulate (secrets)
  - Design decisions that are closely related (likely to change together , high cohesion) are grouped in the same submodule
  - Design decisions that are weakly related (likely to change independently) are allocated to different modules
  - Characterize each module by its secrets (what it hides)
- Viewed top down, each module is decomposed into submodules such that
  - Each design decision allocated to the parent module is allocated to exactly one child module
  - Together the children implement all of the decisions of the parent
- Stop decomposing when each module is
  - Simple enough to be understood fully
  - Small enough that it makes sense to throw it away rather than re-do
- This is called an *information-hiding decomposition*

Specify the Module Interfaces

- The leaf modules in the hierarchy represent units of work
- For each leaf module, we specify
  - Services: the services the module provides that other modules can use
  - Secrets: implementation and design decisions the module must encapsulate
- We must also write a detailed interface spec. (the contract)

Example: GRR

Basic Requirements

The Game Rules Reader (GRR) provides an engine for presenting game rules (and associated information) to an individual on a computer screen. Minimally, the GRR should provide the capabilities of a basic hyper-text document reader such as:

- **Present the current "page" of a hyper-text manual**
- **Allow paging, scrolling, and other standard navigation between pages**
- **Support hyperlinks from one part of the document to another**
- **Support highlighting text**
- **Support writing and attaching notes to the text**
- **Support minimal customizing of the presentation****

****The basic GRR should support simple customizing of the reading experience. The person writing the rules (we'll call a "Writer") can indicate at least two classes of users. The Writer can then designate whether any given part of the document will or will not appear for the indicated class of user.**

Questions?