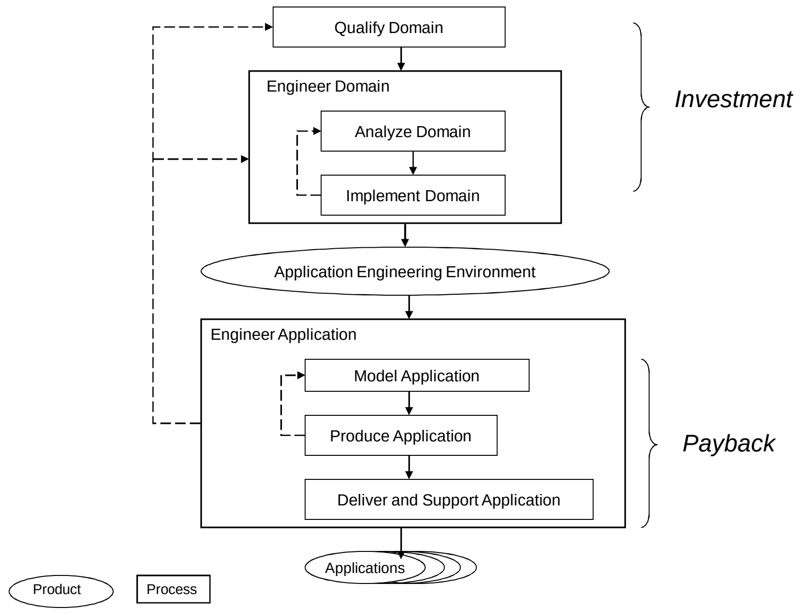
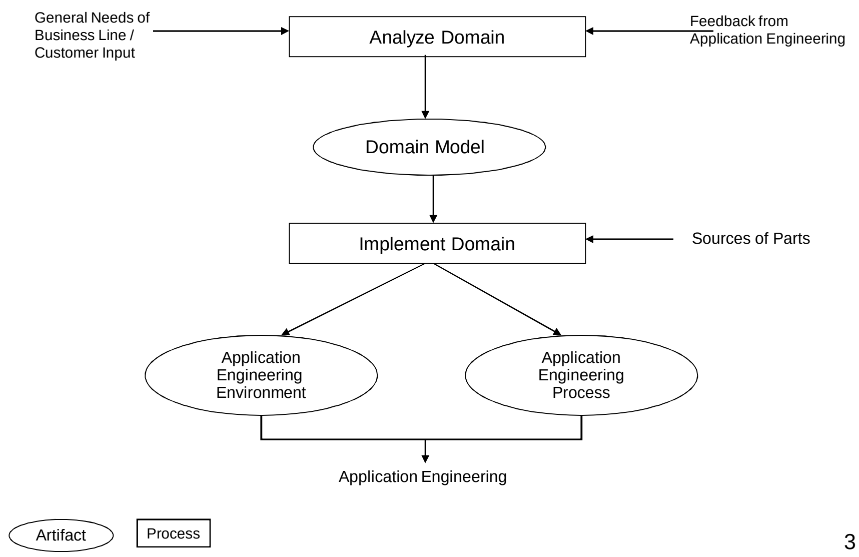
OMSE 551: Week 7
Domain Engineering

- FWS CA
- Domain Engineering and PL Architecture
- Projects

FAST Process Pattern



Ideal Domain Engineering Process


General Needs of Business Line / Customer Input
Analyze Domain
Feedback from Application Engineering
Domain Model
Implement Domain
Sources of Parts
Application Engineering Environment
Application Engineering Process
Application Engineering
Artifact
Process

Analyze Domain

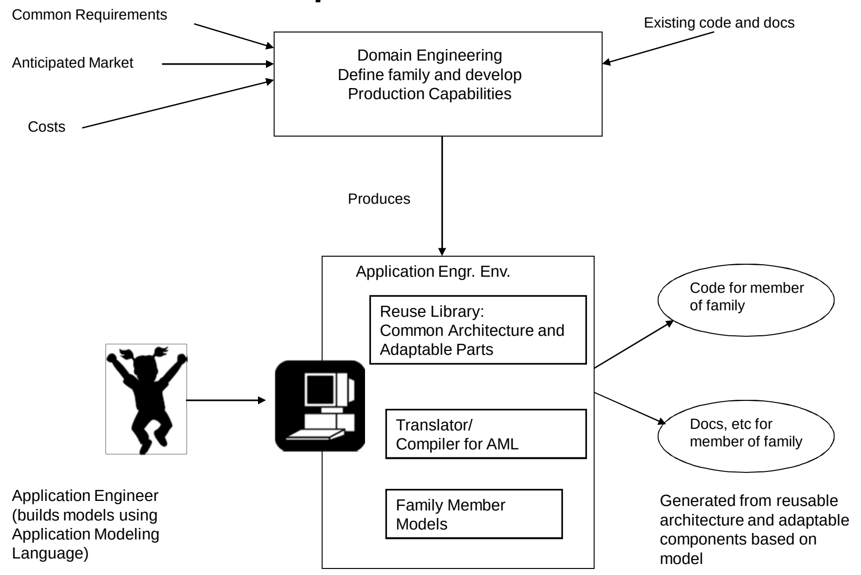
- Process
  - Analyze Commonalty
  - Define Decision Model
  - Design Domain
  - Design AML
  - Design Application Engineering Environment
  - Create Standard Application Engineering Process
- Product
  - Commonalty Analysis
  - Decision Model
  - Family Design
  - Composition Mapping
  - Application Modeling Language
  - Toolset Design
  - Application Engineering Process

Implement Domain

- Process
  - Implement Application Engineering Environment
  - Document Application Engineering Environment
- Product
  - Library: Library of code, documentation and other family artifacts
  - Generation Tools: Tools for generating an instance of the family.
  - Analysis Tools: Tool for analyzing properties of an application model or artifact
  - Documentation: Documentation of how to use the application engineering environment.

5

Graphical View



6

FAST Family Design

- How do we get from CA to a design supporting a product line?
- Focus on developing 1) common architecture and 2) adaptable components
  - Called the “composer” approach
  - Will discuss “compiler” (e.g, compiler compilers) approach later
- Design fundamentally a decision making process
- Family design is an exercise in delayed binding of particular decisions (which?)
- Design principles are key to effective ordering
  - Role of information hiding?
  - Role of abstraction?
  - Role of most-solid-first?

7

Review of Architectural Concepts

8

Working Definition

"The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them."

From *Software Architecture in Practice,* Bass, Clements, Kazman

9

Architectural Structures

- To describe one, we must answer:
  - What are the components?
  - What are the relations?
  - What are the interfaces?
  - To design one, must also know what it is good for - i.e., what quality attributes are affected by that structure.
- Common among views (abstractions!!):
  - Each portrays some aspects of the system, abstracts from others
  - Each supports design of some system attribute (or related set) important to some stakeholder
  - Exactly what is visible and what is abstracted away depends on which attributes must be engineered
  - Which attributes do we want to support here?

10

Architectural Design

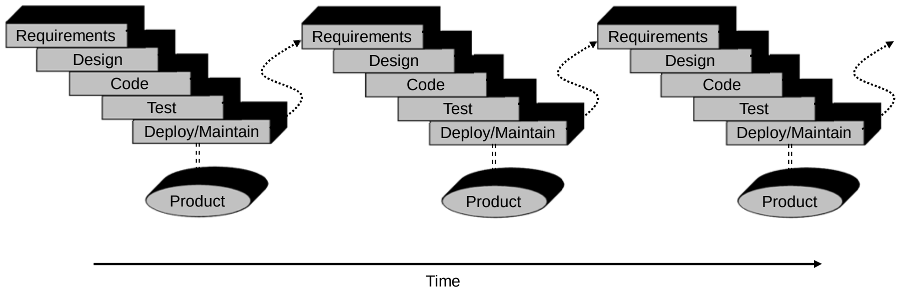
- What does “design an architecture” imply?
  – That we design to a purpose
    - Control of *architectural properties*
  – That we need to choose appropriate components and relations for the purpose
  – That we need a decomposition method (design approach)
    - Principles to guide making appropriate choices
    - Representation to make it visible
  – That we need some metric of “goodness” relative to purpose

11

Designing a Family Architecture

12

Sequential Development Over Time



… a result of “tactical software engineering”

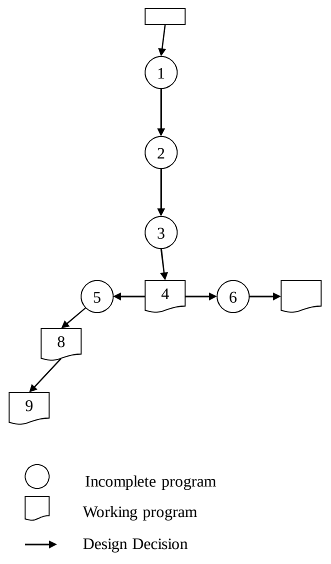
13

Inefficiencies of Sequential Development

- Hypothesis: much of software development is re-development.
  - Software inevitably exists in many versions
  - Seldom develop truly new applications
- Implication: typically much in common among our systems …But very little is reused
  - Difficult to identify commonalties and differences
  - Difficult to reuse code components
  - Difficult to add desired feature to existing design
  - Difficult to adapt other work products (if they exist)
  - Generally easier to re-do than re-use
- What makes work products difficult to reuse?

14

Sequential Development

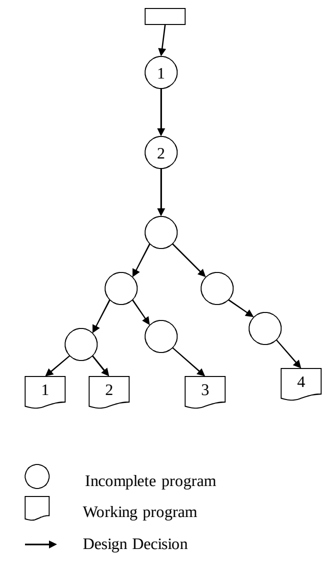


- Built-in dependencies make conceptual structures difficult to reuse

From Parnas: On the Design and Development of Program Families

15

Family Development Model



- Development of new system begins from an intermediate stage
  - Order of decisions is critical
  - Intermediate representation is important
- Order decisions so those common to family occur first
  - Which principle is being applied here?
- Later decisions depend only on those unlikely to change (common to the family)

From Parnas: On the Design and Development of Program Families

16

Role of Architecture

- Architecture provides the basis for making and representing common design decisions for a family
  - Architecture defines: decomposition into parts, relationships between parts, and externally visible component behavior
  - Use architecture to capture common design elements
  - Architectural representation provides the “intermediate representation” for the family
  - Instances of a family share common architecture, differ in implementation details
  - Common architecture, adaptable components, permits rapid creation of members
- How do we systematically construct one?

17

Development Approach

- Goals:
  - Develop a common, reusable architecture for a software product line
  - Develop parameterized, reusable modules (and docs.)
  - Define process for constructing instances
- Approach: reorganize the development process
  - Develop common architecture as distinct product
  - Use common architecture as basis for creating product-line members
- Inputs
  - Domain information (customer intelligence) about expected variations among members
  - Domain information about common assumptions

18

Family Architecture Design

- Approach used in FAST text (based on SCR)
- How is the system decomposed into parts?
  - System is decomposed into a hierarchy of information-hiding modules.
  - Structural decisions common to the family members are made first – result in overall structure and interfaces
  - Variabilities are pushed to the leaf modules when possible

19

Examples

A7E Module Structure

FWS

20

Guide Decomposition

- Approach: structure the modules into classes (taxonomy)
- Use hierarchical structure guided by IH
  - Criterion: System details likely to change independently should be encapsulated in different modules
  - Decompose top-down with categories of change
  - Characterize each module by its secret (what it hides)
- Decompose each module into submodules
  - The children implement the secret of the parent
  - Document the relation in the module guide

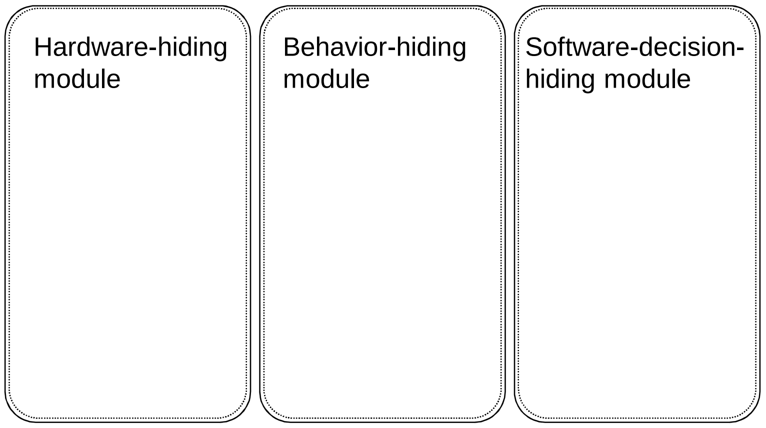
21

Classifying Changes

- Three classes of change
  - hardware
    - new devices
    - new computer
  - required behavior
    - new functions
    - new rules of computing cockpit displays
    - new modes
  - software decisions
    - new ways to schedule processes
    - new ways to represent data types
    - new ways to keep data current

$\left.\begin{matrix} \text{hardware} \\ \text{required behavior} \end{matrix}\right\}$ From Requirements Specification

Three First-Level Modules (Work Assignments)



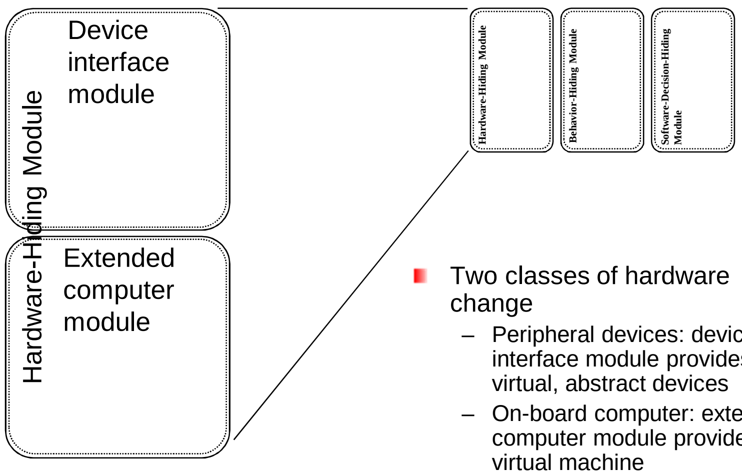
Based on CMU/SEI tutorial for Software Architecture in Practice

Level 1 Documentation

- A:I HARDWARE-HIDING MODULE
  - The programs that need to be changed if any part of the hardware is replaced by a new unit with a different hardware/software interface but with the same general capabilities. This module implements virtual hardware that is used by the rest of the software.
  - Primary secret: the hardware/software interfaces described in chapters 1 and 2 of the requirements
  - Secrets: the data structures and algorithms used to implement the virtual hardware.
- A:2 BEHAVIOR-HIDING MODULE
  - Programs that need to be changed if there are changes in the sections of the requirements document that describe the required behavior.
  - Primary secret: The content of those sections. These programs determine the values to be sent to the virtual output devices provided by the Hardware-Hiding Module.

24

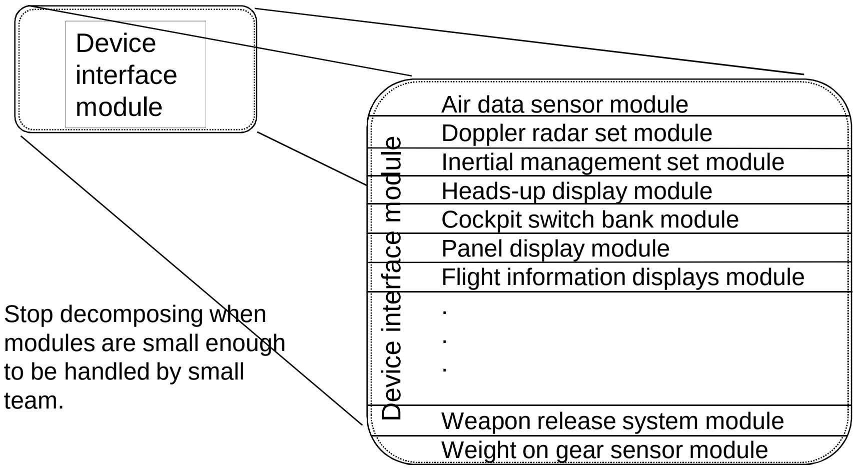
Hardware-Hiding Modules



- Two classes of hardware change
  - Peripheral devices: device interface module provides virtual, abstract devices
  - On-board computer: extended computer module provides virtual machine

Based on CMU/SEI tutorial for Software Architecture in Practice

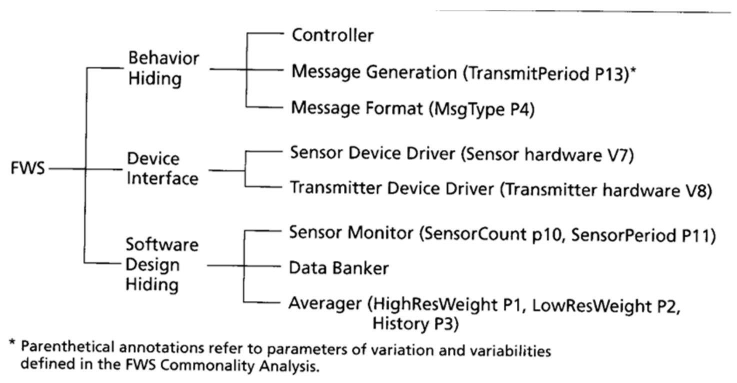
Example of Third-Level Modules


Device interface module
Device interface module
Air data sensor module
Doppler radar set module
Inertial management set module
Heads-up display module
Cockpit switch bank module
Panel display module
Flight information displays module
.
.
.
Weapon release system module
Weight on gear sensor module

- Stop decomposing when modules are small enough to be handled by small team.

Based on CMU/SEI tutorial for Software Architecture in Practice

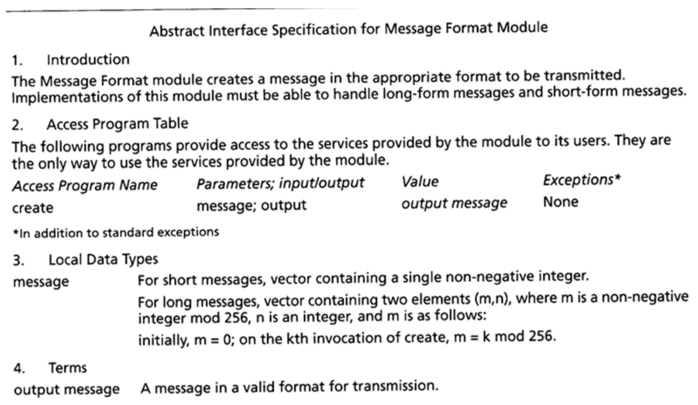
Compare to FWS Architectures


FWS
Behavior Hiding
Device Interface
Software Design Hiding
Controller
Message Generation (TransmitPeriod P13)*
Message Format (MsgType P4)
Sensor Device Driver (Sensor hardware V7)
Transmitter Device Driver (Transmitter hardware V8)
Sensor Monitor (SensorCount p10, SensorPeriod P11)
Data Banker
Averager (HighResWeight P1, LowResWeight P2, History P3)
* Parenthetical annotations refer to parameters of variation and variabilities defined in the FWS Commonality Analysis.

- Reuse of architectural structures
- Variabilities pushed to leaves of hierarchy

27

FWS Architectures



Abstract Interface Specification for Message Format Module

1. Introduction

The Message Format module creates a message in the appropriate format to be transmitted. Implementations of this module must be able to handle long-form messages and short-form messages.

2. Access Program Table

The following programs provide access to the services provided by the module to its users. They are the only way to use the services provided by the module.

Access Program Name	*Parameters; input/output*	*Value*	*Exceptions**
create	message; output	*output message*	None

*In addition to standard exceptions

3. Local Data Types

message	For short messages, vector containing a single non-negative integer.
	For long messages, vector containing two elements (m,n), where m is a non-negative integer mod 256, n is an integer, and m is as follows:
	initially, m = 0; on the kth invocation of create, m = k mod 256.

4. Terms

output message	A message in a valid format for transmission.

- Modules defined using abstract interfaces
- Hide likely changes (variabilities)

28

FWS Uses

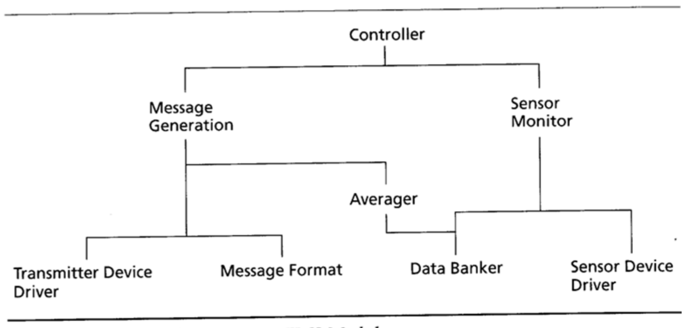


Figure 5-9. *Uses Relation among FWS Modules*

- Uses tells us which modules must be present in a working system
- i.e., if A uses B, both must be included

29

P/L Architecture Design

- Module decomposition
  - Decompose according to *information hiding principle* and *uses relation*
  - Decisions that vary from one family member to the next are “hidden” if possible
    - Goal is to make variable parts easy to change
    - Appear in the leaf-level modules an/or hidden structures
    - Alternatives are parameterized when possible
  - Uses relation is designed to be “loop free”
    - Programs at lower levels in the module hierarchy may no use programs in the modules above them
    - Programs at the top level may only use each other
    - Consistent with layered structure though layering may not be strict
- Evaluate according to ease of creating expected family members

30

Organizing Principles

- Information Hiding
  - Variabilities confined to single modules or module interior
    - Replace a module implementation
    - Parameterize a module implementation or interface
  - Commonalties eligible for architecture (component structure and interfaces)
- Abstraction
  - Architecture to family members is many-to-one
  - Looking for characteristics that are true of all family members.
- Uses
  - Defines what’s needed for any subset
  - Used by generation mechanism
  - Allows adding/removing programs or components

31

Adaptable Components

32

Goals

- Represent any number of similar software components with a single definition
- Mechanically retrieve/generate a particular component instance by resolving deferred decisions
- Cost no more to develop than 1-3 individual components

Definitions

- Component: A fragment of a work product
- Component Family: A set of components that are sufficiently similar to be described effectively by the same abstraction
- Abstraction: A concept that characterizes any instance of a family equally well
- Metaprogram: A program that generates instances of a component family
- Adaptable component: A representation of a family sufficient to specify a corresponding metaprogram

The Role of Decisions

- A characteristic set of deferred decisions distinguish among the members of a family.
- An Adaptable Component shows how different ways to resolve a set of decisions lead to different programs.
- Decisions represent:
  - Customer requirements (needs and constraints).
  - Engineering tradeoffs (such as cost, quality, ease of change, esthetics, and feasibility).
- A focus on similar problems (a family) enables standardization, reducing number, variety, and complexity of decisions.

Keys to Reuse Success

- Standardization: Avoid incidental differences between similar reusable components.
- Easy (transparent) customization: Accommodate essential differences needed to satisfy specific needs.
- Ownership: Guarantee that somebody knows how each component works and is responsible for error fixes and enhancements.
- Motivation: Create reusable components based on expectations about future needs.

Parts of an Adaptable Component

- An abstraction: What is the intended purpose of these components?
- Parameters (representing decisions): Why is there a need for more than one of these components? How are they different from each other?
- A definition: Given a set of parameter values, what are the steps to create a corresponding component?

Precursor Mechanisms

- Alternative implementations of standardized components
- Generalized (runtime-adaptive) components
- Partial-code generators (GUI, parsers, etc.)
- Word processor conditional/form letter mechanisms
- Compiler macros, flags, and switches
- Object-oriented language mechanisms: subclasses, inheritance
- Templates (C++)/generics (Ada)

Motivations for a Special-Purpose Mechanism

- A set of similar components can be concisely represented in one unified source.
- Form and content of instances is easily perceived.
- Adaptations are traceable entirely to parameters.
- Parameters can be expressed at a problem-level, independent of solution details.
- Instance components can be derived mechanically.

39

Basic Mechanisms

- *MetaProgram Instantiation*: indicates a point within the output text for a metaprogram instance at which the metaprogram's body should be expanded and included.
- *Value Substitution*: a point at which the value of a referenced slot or generator should be included.
- *Value Selection*: a point at which one of several alternative text fragments should be expanded and included.
- *Value Repetition*: a point within the output text at which zero ofrmore concatenated instances of a text fragment should be expanded and included.

40

Fixed Size, Type Stack

```
public class intStack {

    static final int maxSize = 1024;
    int data [] = new int [maxSize];
    int size = 0;

    public void add (int p1) throws stackFull {
        if (size == maxSize) throw new stackFull ();
        data [size++] = p1;
        }

    public int get () throws stackEmpty {
        if (size == 0) throw new stackEmpty ();
        return data [--size];
        }
}
```

41

Fixed Size, Variable Type

```
« program stacks (name:text, datatype:text, maxsize:text ) «
public class «name»Stack {

    «datatype» data [] = new «datatype» [«maxsize»];
    int size = 0;

    public void add («datatype» p1) throws stackFull {
        if (size == «maxsize») throw new stackFull ();
        data [size++] = p1;
        }

    public «datatype» get () throws stackEmpty {
        if (size == 0) throw new stackEmpty ();
        return data [--size];
        }
}
» »
```

Metaprogram

Value Substitution

42

Summary

- Family design is an exercise in systematically delaying binding of certain decisions
- Our design principles support developing a design where:
  - Common (most solid) decisions are bound first
  - Variable decisions are bound last
- Information hiding design is a means for implementing such bindings
  - Common decisions in architecture
  - Variable decisions in adaptable components

43

Assignment

- Projects – revise proposals if needed
- Continue with FWSOS module guide
- Will post common CA document

44

Questions?

45