Reusable Components

Overview
- Specifications provide user-oriented cover story
- Designs address efficiency and sufficient functional completeness issues
- Specifications hide implementation-specific information
- Multiple implementations may be developed to satisfy the same specification

Languages for Formal Specification
- ANNA (and SPARK) for Ada
- JML for Java
- Larch/C++ for C++
- Spec# for C#
- ...
- Eiffel
- RESOLVE
- ...
- VDM
- Z

Common Principles: Data Abstraction Specification
- Specifications are contracts
- Formal; unambiguous
- Mathematical logic
- Use a combination of well-known mathematical theories and notation
- Specify mathematical models for objects
- Specify the behavior of operations using those models

Example: Use of Mathematical Theories

Concept Stack_Template(type Entry; ...
- uses String_Theory;

Type Family Stack is modeled by ...

Operation Push...
Operation Pop...
...
end Stack_Template;

Mathematical Modeling: Recall IntStack Interface
- Suppose IntStack is an interface
  - uses Integer_Theory, String_Theory;
- Think of stacks of Integers as “math strings” of integers
  - this: Str(Z);
- Suppose Max_Depth is the maximum size
  - Constraints |this| <= Max_Depth;
- Specification of Constructor
  - Initialization ensures this = empty_string;
- Exercises: Specification of other Stack operations
Mathematical Modeling

**Concept** Stack_Template(type Entry;
Max_Depth: Integer);
requires Max_Depth > 0;
uses String_Theory;

**Type Family** Stack is modeled by Str(Entry);
exemplar S;
constraints |S| <= Max_Depth;
initialization
   ensures S = empty_string;

---

Specification of Operations: Recall IntStack Operations

- **Operation** push (int x);
  - updates this; restores x
  - requires |this| < Max_Depth
  - ensures this = <x> o #this;
- int Operation pop ();
  - updates this;
  - requires |this| > 0;
  - ensures #this = <result of pop> o this;
- int Operation depth ();
  - preserves this;
  - ensures result of depth = |this|;

---

Specification of Operations

- Operation Push (alters E: Entry; updates S: Stack);
  - requires |S| < Max_Depth;
  - ensures S = <#E> o #S;
- Operation Pop (replaces R: Entry; updates S: Stack);
  - requires |S| > 0;
  - ensures #S = <R> o S;
- Operation Depth (restores S: Stack): Integer;
  - ensures Depth = |S|;

---

Using Reusable Components

- Users (clients) need to know only interface specifications
- Users need to supply appropriate parameters to instantiate
- Depending on the paradigm, special operations are automatically available on objects
  - Assignment in Java (e.g., S = T)
  - Swap in RESOLVE (e.g., S :=: T)

---

Multiple Implementations

- Alternative implementations provide the same functionality
- Provide performance trade-offs
  - time vs. space
  - average case vs. worst case
  - Efficiency vs. predictability
  - some subset of methods vs. some others
- Users pick ones best fitting their requirements when instantiating

---

Common Principles: Data Abstraction Implementation

- Identify data representation (e.g., use of arrays or vectors to store Stack contents)
- Specify representation invariants
- Specify correspondence between representation and abstraction
- Write code for each operation to meet specifications
**Example Data Representation**

```java
public class Array_Realiz<Entry>
    implements Stack_Template<Entry>
{
    private Entry contents[];
    private int top;
    ...
    // methods
}
```

**Realization**

```java
Array_Realiz for Stack_Template;
Type Stack = Record
    Contents: Array (0..Max_Depth-1) of Entry;
    Top: Integer;
end;
...
end Array_Realiz;
```

**Which Code for Push is Right?**

- **Procedure Push**
  ```java
  Procedure Push(alters E: Entry; updates S: Stack);
      S.Top++;
      S.Contents(S.Top) :=: E;
  end Push;
  ```

- **Procedure Push**
  ```java
  Procedure Push(alters E: Entry; updates S: Stack);
      S.Contents(S.Top) :=: E;
      S.Top++;
  end Push;
  ```

- **Procedure Push**
  ```java
  Procedure Push(alters E: Entry; updates S: Stack);
      S.Top--;
      S.Contents(S.Top) :=: E;
  end Push;
  ```

**Internal Contracts (Part 1)**

- **Convention or representation invariant**
  - All public methods (except constructors) may assume it at the beginning
  - All public methods must guarantee it at the end
- **Examples:**
  - Conventions 0 <= S.Top <= Max_Depth – 1;
  - Conventions 1 <= S.Top <= Max_Depth;
- **Which constructor (initialization) code is OK?**
  - S.Top := 0;
  - S. Top := 1;
  - S.Top := -1;
  - S.Top := Max_Depth;

**Internal Contracts (Part 2)**

- **Correspondence or abstraction relation**
  - Relationship between the internal representation and the conceptual (or abstract) model in the specification
  - Holds when the conventions hold
  - Examples:
    - Correspondence Conceptual S = the entries in array S.Contents from 0 to S.Top -1 in reverse order
    - Correspondence Conceptual S = the entries in array S.Contents from S.Top to Max_Depth -1 in the same order
- **Which constructor (initialization) code is OK?**
  - S.Top := 0;
  - S. Top := 1;
  - S.Top := -1;
  - S.Top := Max_Depth;

**Internal Contracts**

- **Write initialization (constructor), push, and pop code that satisfy the following internal contract**
  - Conventions 0 <= S.Top <= Max_Depth – 1;
  - Correspondence Conceptual S = the entries in array S.Contents from 0 to S.Top -1 in reverse order
Exercises and Questions

- Write code using different conventions and correspondence assertions
- Combine code for different methods from different people
- Does it work? Why or why not?