Prosperity
Heights
Software

@

Adaptable Components
- for Flexible Reuse -

August 25, 2000

Grady H. Campbell, Jr.

A Usual Context for Reuse

- **Writing a program that is somewhat similar to past programs.**
- **The program is organized into a set of "components" for modularity.**
- **Each component has a specified interface that other components reference.**
- **Each component can be**
  - **written from scratch**
  - **reused, perhaps with changes, from past work.**

When would reuse be the right choice?

A Conventional View of Reuse

1. **Find previously built components similar to what you need.**

2. **Choose one that best matches what you need.**

3. **Change it so it that it does exactly what you need.**

4. **Save it for future reuse by others?**

Questions with this Approach

- Does the component you need exist and can you find it?
- Alternatively, do components similar to what you need exist? Which one will be easiest to change to fit your needs?
- Does the component you need work correctly? If you have to change it, will it still work correctly?
- Does the component do things you don't want? Can you safely remove them?
- How long will all of this take and wouldn't it be easier just to write it yourself?

Analysis

- **Reuse ought to be routine for a reliable, cost-effective software development process.**

- **A conventional approach to reuse**
  - **is problematically opportunistic.**
  - **makes the reuser do most of the work, within poorly specified limits assumed by the developer.**
  - **puts all risk on the reuser, without institutional support. (“reuse to save effort but if it doesn’t work out it’s your problem”)**
  - **never establishes why similar solutions are possible.**

- **A better conceived, less simplistic approach to reuse is needed.**

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.**

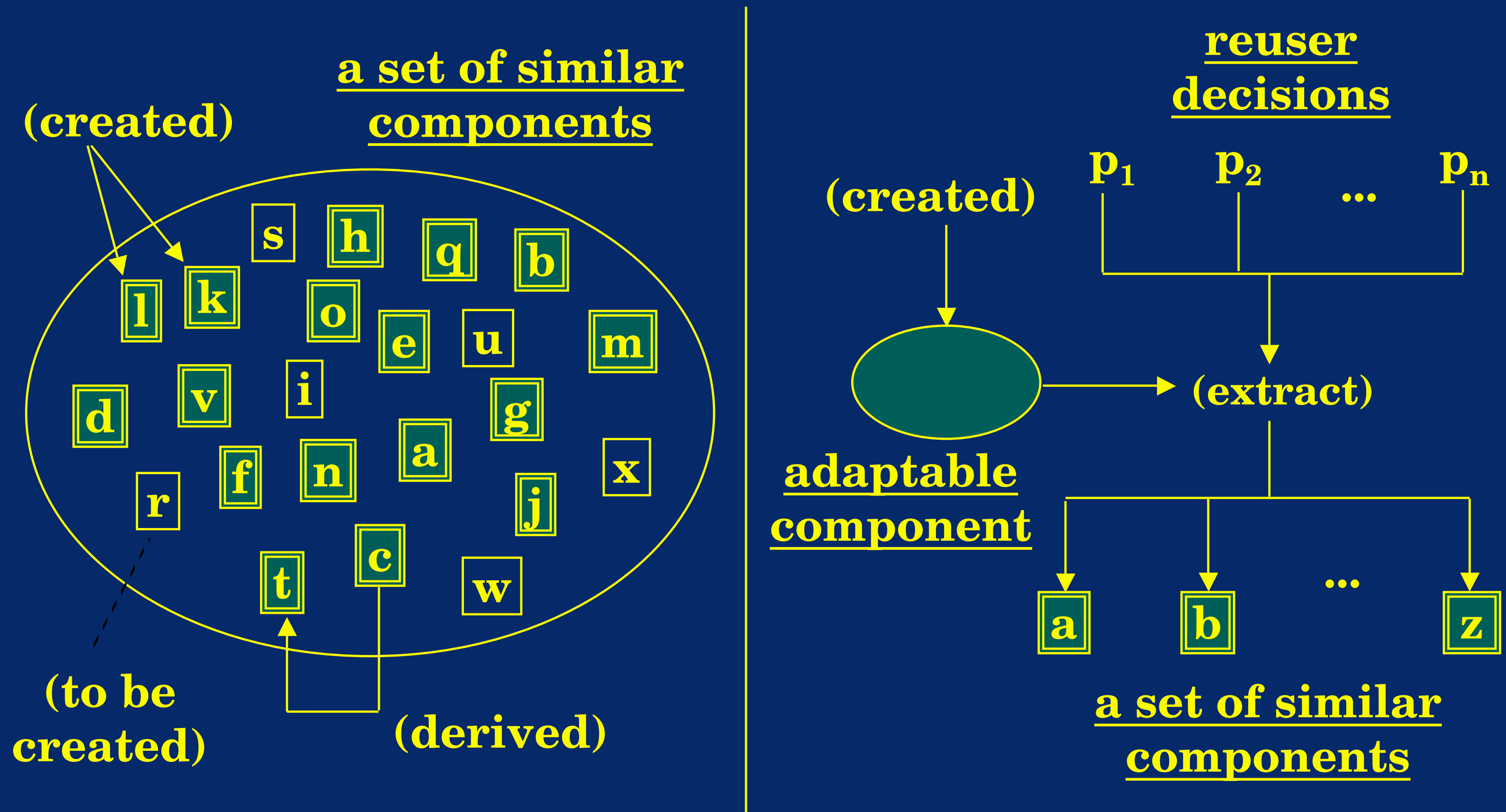
- **Motivation: Create reusable components based on expectations about future needs.**

A Basic Tenet for Systematic Reuse

The only sound basis for reuse is an envisioned set of *similar* products or components: a <u>family</u>.

- **Similarity**
  - **Commonality: the basis for *standardization* (of work products and process)**
  - **Variability: the *flexibility* needed to accommodate different needs**
- **Adaptability**
  - **An explicit representation of similarity**
  - **A characteristic set of deferred <u>decisions</u> that distinguish among the members of a family**

2 Views of Reusable Components



Nature of an Adaptable Component

- Definition: A family of similar components
- Purpose: Supply customized reusable components
- Parts:
  - An abstraction: What is the intended purpose of these components?
  - Parameters (reuser 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?

The Role of Decisions

- Engineering is a decision-making process.
- 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.

Motivations for Adaptable Components

- Adaptable components support diversity and change:
  - Effective reuse requires tailoring to specific needs
  - Tailoring is decision-based and mechanical
- Repository-associated costs are minimized:
  - Developer builds one component for multiple needs
  - Storage space is a fraction of storing an equivalent set of instance components
  - Reuser effort and risks are reduced
- Maintenance of one Adaptable Component is easier:
  - Errors are fixed once
  - Improvements are available to all

References

- **E. W. Dijkstra, "On Program Families",** ***Structured Programming*****, Academic Press, London, 1972, 39-41.**
- **D. L. Parnas, "On the Design and Development of Program Families",** ***IEEE Trans. Software Eng*****. SE-2 (March 1976), 1-9.**
- **J. A. Goguen, "Parameterized Programming",** ***IEEE Trans. Software Eng*****. SE-10, 5 (September 1984), 528-543.**
- **N. Dershowitz, "Program Abstraction and Instantiation",** ***ACM Trans. Prog. Languages & Systems*** **7, 3 (July 1985), 446-477.**

For more information: <www.domain-specific.com>

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 easy to envision.
- All tailoring is traceable entirely to parameters.
- Parameters can be expressed at a problem-level, independent of solution details.
- Instance components can be derived mechanically.

A.C. Example
Sequenced Collections

A progression from specific to abstract:

1. **Fixed-size, fixed-type stack**

2. **Fixed-size, variant-type stack**

3. **Variant-size, variant-type stack**

4. **Variant-size, variant-type, variant-access sequence (stacks, queues, deques)**

F-size, F-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];
        }
}
```

F-size, V-type Stack

```
« 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];
            }
}
» »
```

V-size, V-type Stack

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

    «maxsize?««datatype» data [] = new «datatype» [«maxsize»]; int size = 0»
        : «Vector data = new Vector () »»;

    public void add («datatype» p1) {
        data«maxsize?« [size++] = p1»:«.put (p1)»»;
        }

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

}
» »
```