Prosperity Heights Software

ICSE 99 Tutorial

Adaptable Components

May 18, 1999

Grady H. Campbell, Jr.

Goals for the Tutorial

- Explain the concept and motivation of Adaptable Components
- Present alternate mechanisms for representing an Adaptable Component
- Show how to create an Adaptable Component:
  - by abstracting from a single-use component
  - by unifying similar existing components
- Look beyond, to Domain-specific Engineering

Objectives of Reuse

- Produce a product
  - with better quality
  - in less time
  - using less detailed expertise
- Leverage an organization's combined knowledge and expertise.
- Provide a rapid prototyping capability:
  - develop better understanding of a customer's needs
  - explore alternative solutions to a customer's problem

What Does Not Work

- **A library containing thousands of single-use parts**
- **Text-based searching for a needed part**
- **Incentives to write or use 'reusable' parts more**
- **Generic parts, beyond current use**
- **Reworking a single-use 'reusable' part to suit a new single-use need**

What Does Work

Adaptable Components in a product line context

- **Reusable parts built for multiple use (tailorable to different needs)**
- **Reuse based on an analysis of current and likely future needs, viewing differences as deferred decisions**
- **Systematic reuse as a product line investment strategy**

Goals for Adaptable Components

- **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**

Topic Outline

- **What is an Adaptable Component?**
- **Notations, General and Special**
- **Examples**
- **Exercise**
- **Summation**

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**

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 money 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.**

Foundations

- **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.**

A Basis for Effective Reuse

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

2. Similarity implies both *commonality* and *variability:*
   - Commonality is the basis for *standardization* (of work products and process – for a domain).
   - Variability characterizes the *flexibility* needed to accommodate different needs.

3. *Adaptability* is an explicit representation of similarity:
   - A characteristic set of deferred decisions distinguish among the members of a family.

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.**

Topic Outline

- What is an Adaptable Component?
- Notations, General and Special
- Examples
- Exercise
- Summation

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?**

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.**

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.**

An Adaptable Component Notation

- Target text (common parts of components)
- MetaPrograms (variant parts of components)
  - Name, to identify the abstraction
  - Parameters, with which reusers control tailoring
  - Definition (target text containing metaConstructs), to show how to extract tailored instances

```
« program F ( p1: text,
         p2: (p3:text*, p4:text)? ) «
   some common text «p1» «
   p2.p3
      ? «repeating text:«for p in p2.p3 ««p»»»
      : «alternate text:«p2.p4»»
      »
   » »
```

Parameters

- A text value referenced by the name p1

  `p1 : text`

- Optional

  `p1 ?: text`

- Symbolic (optional, non-valued)

  `p1 ?`

- Multivalued

  `p1 : text *`

- Structured or variant

  `p1:(p:text*, q:( q1?, q2:(r1:text*, r2:text)? ))`

MetaConstructs

- **Substitution:**
  `« p1 »`
- **Selective substitution:**
  `« p1.q.q1 ? «with q1» : «without q1» »`
- **Repetitive substitution:**
  `« for i in p1.p «same but different due to «i».»»`
- **Metaprogram instantiation:**

```
« F  (  p1: «in all work products»,
        p2: (p3:(«value 1», «value 2», «value 3»))
     ) »
```

Writing an Adaptable Component

1. Write a prototypical instance component.

2. Write a top-level metaProgram based on major decisions.

3. Derive prototypical and 2-3 new instance components.

4. Refine the metaProgram to support extended/refined decisions, based on experience in #3.

5. Extend the metaProgram based on likely future needs.

6. Write subordinate metaPrograms to manage complexity.

7. Regenerate old instances to verify and update, as needed.

Topic Outline

- **What is an Adaptable Component?**
- **Notations, General and Special**
- **Examples**
- **Exercise**
- **Summation**

Example Adaptable Components

- **Sequenced collections**
- **Application-specific spreadsheets**
- **Specialized reuser tools**

A.C. Example 1
Sequenced Collections

A progression from the specific to the 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 ()»»;
        }

}
» »
```

V-size, V-type, V-access Sequence

```
« program lifoProcs (name:text, datatype:text, maxsize:text) «
    public «datatype» getFirst () throws «name»Empty {
        if («maxsize=«»?«size»:«data.size()»» == 0) throw new «name»Empty ();
        return data«maxsize?« [--size]»:«.get ()»»;
        }
» »
. . .
« program sequence (name:text, datatype:text, maxsize?:text, access:(fifo?,lifo?)) «
public class «name» {
    . . .
    public void add («datatype» value) { . . . }

«access.lifo ? ««lifoProcs (name:««name»», datatype:««datatype»»,
    maxsize:««maxsize?««maxsize»»:«»»» »»»
«access.fifo ? ««fifoProcs (name:««name»», datatype:««datatype»»,
    maxsize:««maxsize?««maxsize»»:«»»» »»»
    . . .
}
» »
```

A.C. Example 2
Application-Specific Spreadsheets

- **Properties and functions of spreadsheets**
- **Decision specifications for a family of special-purpose spreadsheets**
- **Implementation in the form of a configurable Java applet**

Steps in Using a Spreadsheet

- **Set up**
  - **Layout, labels, and cell formatting**
  - **Cell content functions and data sources**
  - **Applicable chart, analysis, and report types and formats**
- **Use**
  - **Enter raw data**
  - **Generate charts, analyses, and reports**
  - **Verify results**

Objective of adaptability: Minimize set up by users

Detailed Goals of Adaptability

- **Add domain-specific extensions (formulas, analyses, reports, procedures)**
- **Preverify consistency of system of computations**
- **Remove unneeded capabilities provided in generalized tools**
- **Reduce breadth of required user skills and knowledge**
- **Tailor interface to fit skills and knowledge of a specialized user community**
- **Standardize techniques across a user community**

A Special-Purpose Implementation

- Automate set up to create a spreadsheet tool tailored to a particular user community's needs
- Derived from Sun Java Spreadsheet applet example
- HTML file supplies applet parameter values that guide tailoring
- Example families (subfamilies)
  - Financial recordkeeping (income/expenses, investments)
  - Scheduling (tasks, personnel)
  - Product tracking (orders, in-production, inventory)

Set-Up Decisions

- Spreadsheet title
- Fixed geometry
  - Row and column names
  - Cell names
- Cell content
  - Numeric value
  - Formula (using row@column or cell names)
  - Comment
- Fixed cell content type, fixed computations

Sample HTML Source

```
<applet code="SpreadSheet.class" width=320 height=120>
<param name=title value="Income Statement">
<param name=columnNames
        value="1990,1991,1992, ,Accum,1993, ">
<param name=rowNames value="gross,taxes,net">
<param name=gross@1990 value="10000">
<param name=taxes@1990 value="1600">
<param name=taxes@1991 value="f'gross@1991'*0.22">
<param name=taxes@Accum
        value="f'taxes@1990'+'taxes@1991'+'taxes@1992'">
<param name=net@1990 value="f'gross@1990'-'taxes@1990'">
<param name=net@1991 value="f'gross@1991'-'taxes@1991'">
<param name=net@1992 value="f'gross@1992'-'taxes@1992'">
<param name=taxes@Accum#name value="Prior Taxes">
</applet>
```

Derived Spreadsheet

Income Statement

Prior Taxes: f’taxes@1990’+’taxes@1991’+’taxes@1992’

	1990	*1991*	*1992*		*Accum*	*1993*	
gross	10000	30000	50000			53000	
taxes	1600	6600	9000		**17200**		
net	8400	23400	41000				

A.C. Example 3
Specialized Reuser Tools

- A "generator" Adaptable Component (instances are programs having a graphical interface for instantiating some other Adaptable Component)
- Shown here: reuser interfaces to 2 Adaptable Components whose instances are simple text documents
  - Newspaper Jobs Listing (NJL)
  - Customized Computer Order Invoice (CCOI)

Steps Followed

0. Write the "generator" Adaptable Component (AC_G)

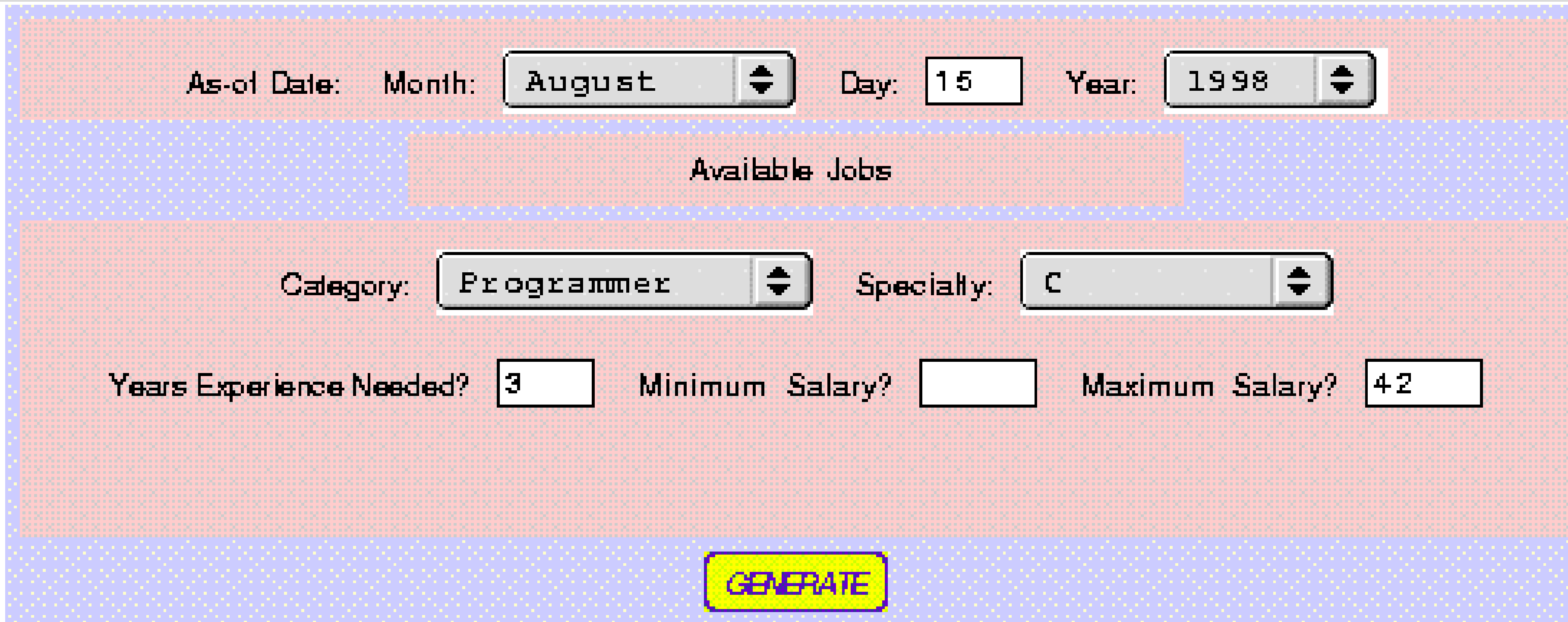
1. Write a "target" Adaptable Component (AC_T) {such as NJL or CCOI}

2. Instantiate AC_G, describing parameters expected by AC_T, to create Java program P_T

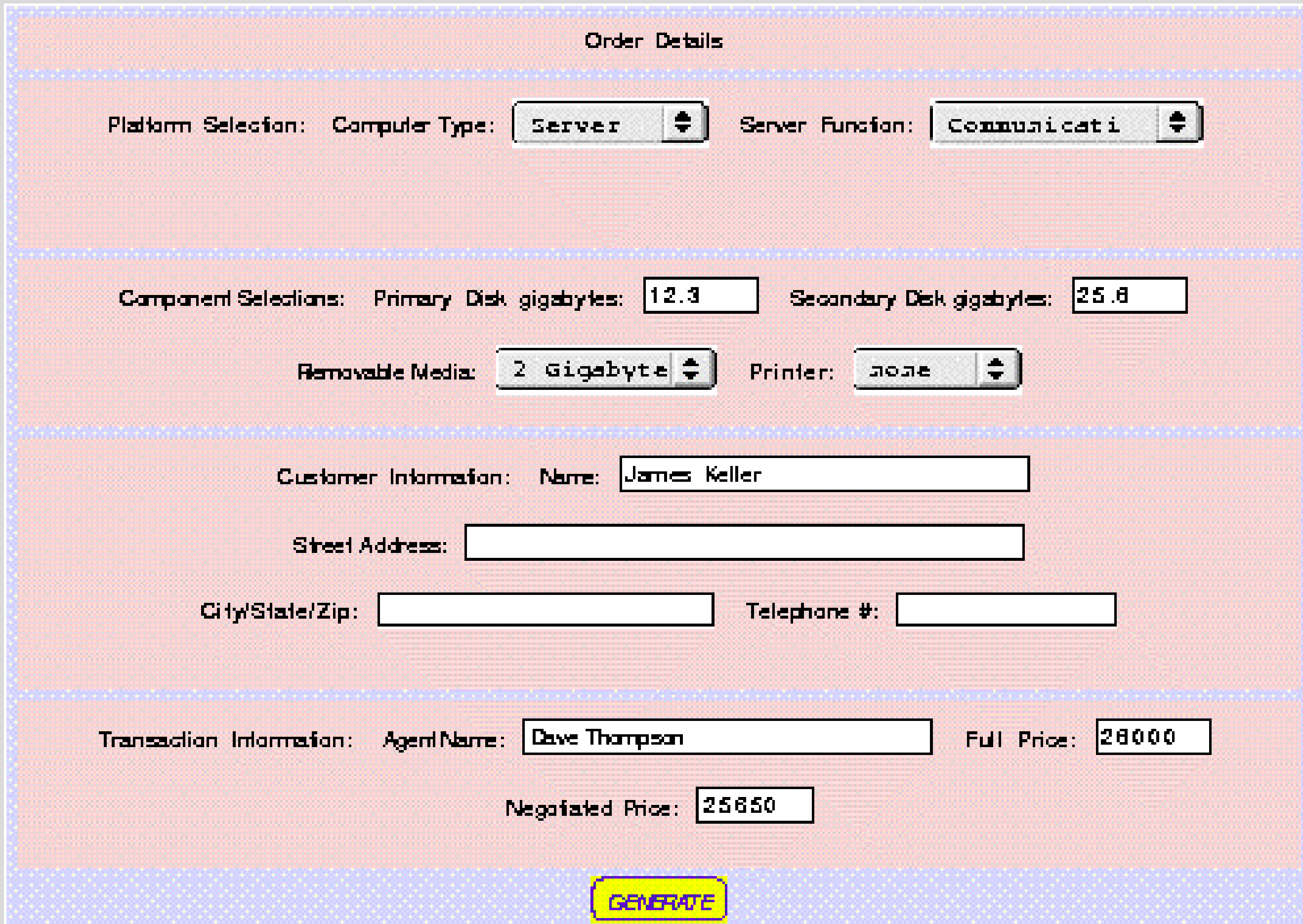
3. Compile P_T

4. Use P_T to
   - input parameter values for AC_T
   - instantiate AC_T

Interface Generated for NJL



Interface Generated for CCOI



Topic Outline

- **What is an Adaptable Component?**
- **Notations, General and Special**
- **Examples**
- **Exercise**
- **Summation**

Exercise Purpose

- **Think about why similar components are different:**
  - **To fill different needs (essential differences)**
  - **Due to different implementers (incidental differences)**
- **Think about how essential differences can be expressed as deferred decisions**
- **Think about how a set of similar components can be represented as an Adaptable Component**

Exercise Procedure

- **Compare sample instances (simple Java code) for similarities:**
  - **Expense ledger**
  - **Job assignment schedule**
  - **Publications reference list**
- **Unify instances to create an informal Adaptable Component (mark up one instance to show how other instances match or differ).**
- **Define an abstraction and deferred decisions for your Adaptable Component (propose a vocabulary for distinguishing the samples as instances of a family).**

Guide to Comparing Instances

- Find similar fragments in any 2 instances:
  - Are there similarities in structure or parts, ignoring incidental differences such as naming?
  - Are there similarities, allowing for consistent essential differences such as data types?
  - What editing actions would make the fragments the same?
  - Do different needs justify differences found?
- See whether each fragment occurs in other instances:
  - Yes, a common element
  - No, other essential or incidental differences

Guide to Unifying Instances

- **Incidental differences: Select best and eliminate alternatives.**
- **Essential differences: Characterize equivalent editing action**
  - **Substitution: replacement with instantiator-provided content, specific to each instance**
  - **Selection: a choice among alternative predetermined contents**
  - **Repetition: repetition, with tailoring, of some standard content**

Guide to Defining an Abstraction

- **Identify a unifying concept:**
  - **Characterize the set of sampled instances**
  - **Generalize to include other likely instances**

- **Identify deferred decisions:**
  - **Characterize decisions that match the abstraction**
    - **Sample-derived decisions may be too solution-specific.**
    - **How would a reuser want to express what they need?**
  - **Group related decisions**
  - **Mark up the Adaptable Component to show where deferred decisions are referenced**

(Do the Exercise)

Questions

- Can any other programs be built with the Adaptable Component you have described? What are they?
- Suppose you wanted to add capabilities to the 3 sample programs. What would you add? Which would be less effort in this case:
  - Modifying each of the sample programs as needed?
  - Extending the Adaptable Component to build the enhanced programs?
- Suppose you found an error in one of these programs. Would it be better to fix that program or to fix an Adaptable Component and regenerate the program? Why?

Topic Outline

- **What is an Adaptable Component?**
- **Notations, General and Special**
- **Examples**
- **Exercise**
- **Summation**

Aspects of Adaptable Components

- **Abstraction of a family of similar instances**
- **Deferred decisions that distinguish among instances**
- **A metaprogram that can generate family instances**
- **Used to retrieve a customized reusable component:**
  - **Make decisions**
  - **Generate instance**
  - **Verify instance for intended use**
  - **Modify decisions, instance, or Adaptable Component, as appropriate**

Motivations for Adaptable Components, Revisited

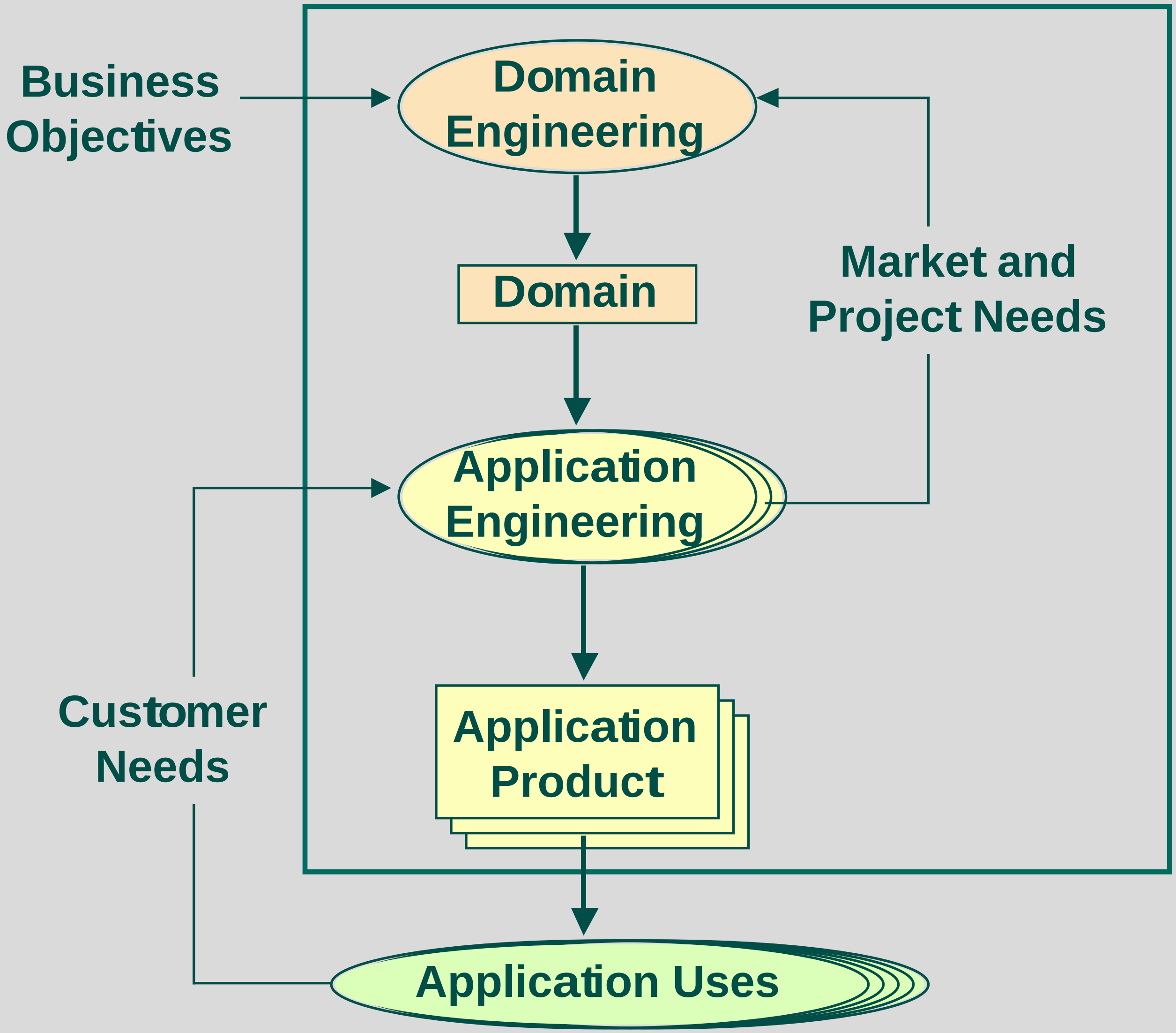
- **Adaptable components support diversity and change:**
  - **Effective reuse requires tailoring to specific needs**
  - **Tailoring is decision-based and mechanical**
- **Repository-associated costs are minimized:**
  - **Developer builds one component for multiple needs**
  - **Storage space is a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

Beyond Adaptable Components: Domain-specific Engineering

Standardization of the most effective solutions to a class of similar problems

- **Identify a product line business area whose customers need similar products.**
- **Develop a shared understanding of how and why needed products are similar.**
- **Create the means to produce standardized, customized products rapidly.**
- **Transition systematically, with tailoring and incremental improvement.**

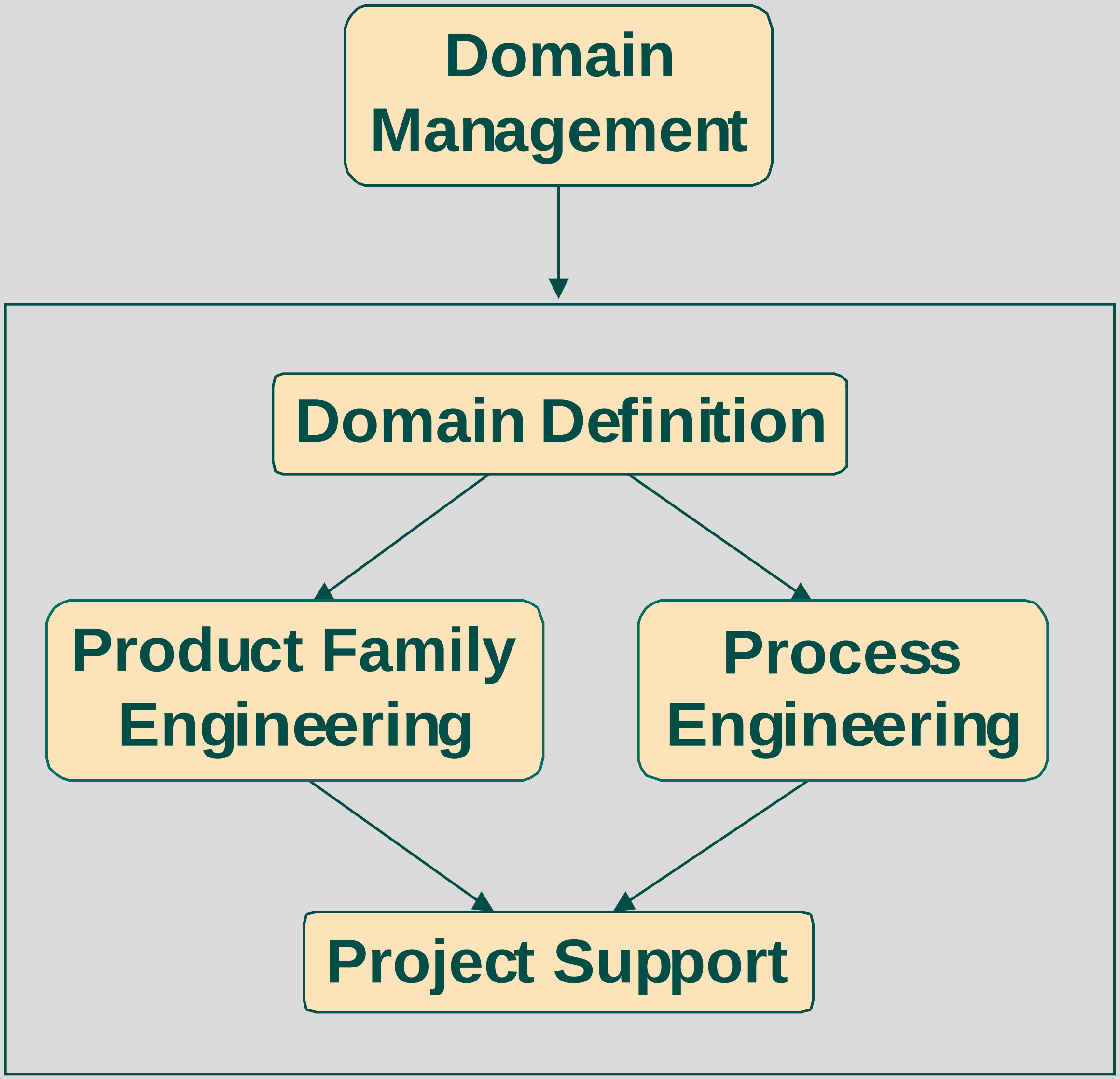
The DsE Process



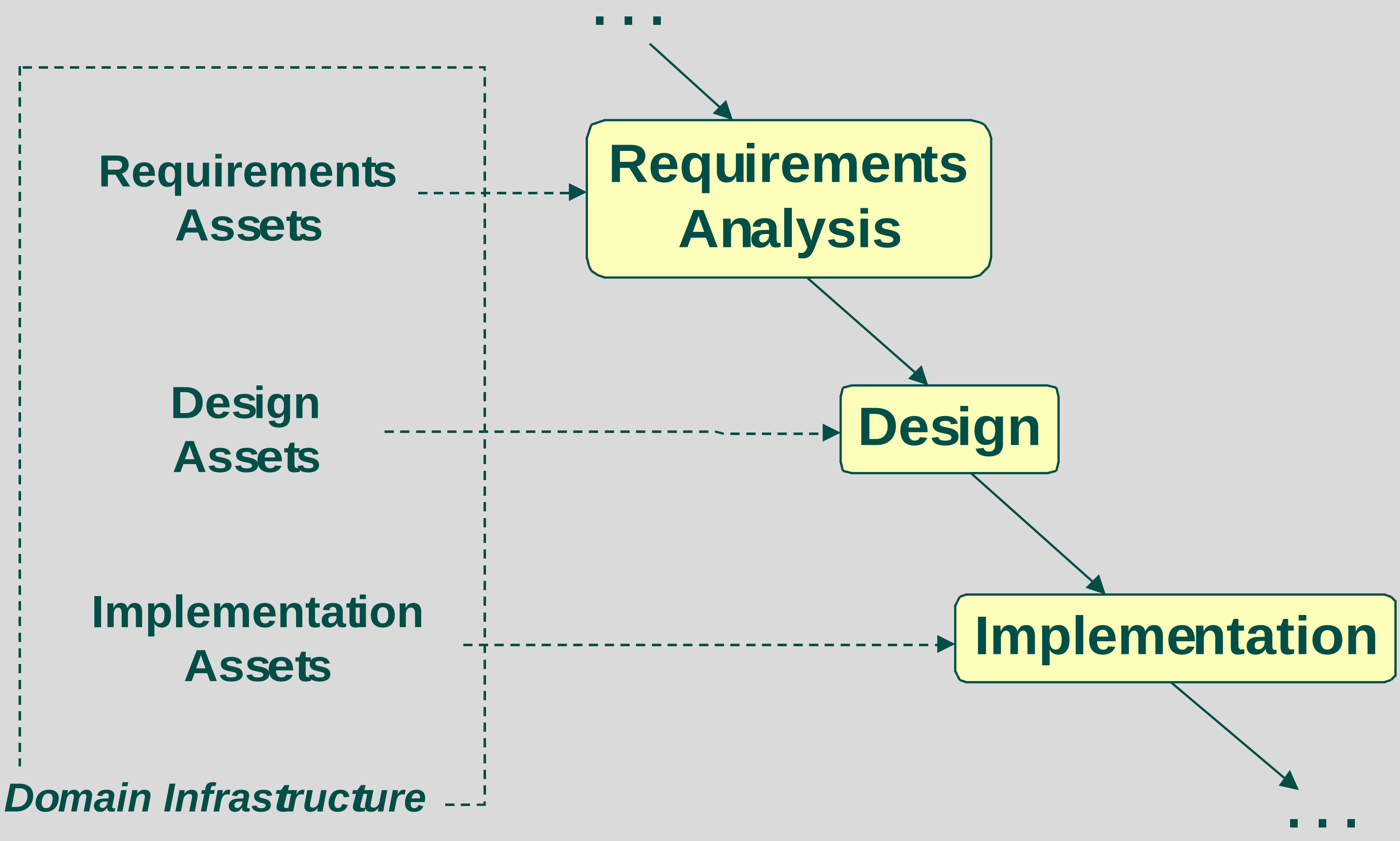
DsE Activities

- **Domain Engineering:**
  - **Standardize a product family, adaptable to deferred requirement and engineering decisions.**
  - **Establish a standard process for resolving deferred decisions.**
- **Application Engineering:**
  - **Resolve deferred decisions to match customer needs.**
  - **Mechanically produce a product, adapted to resolved decisions.**

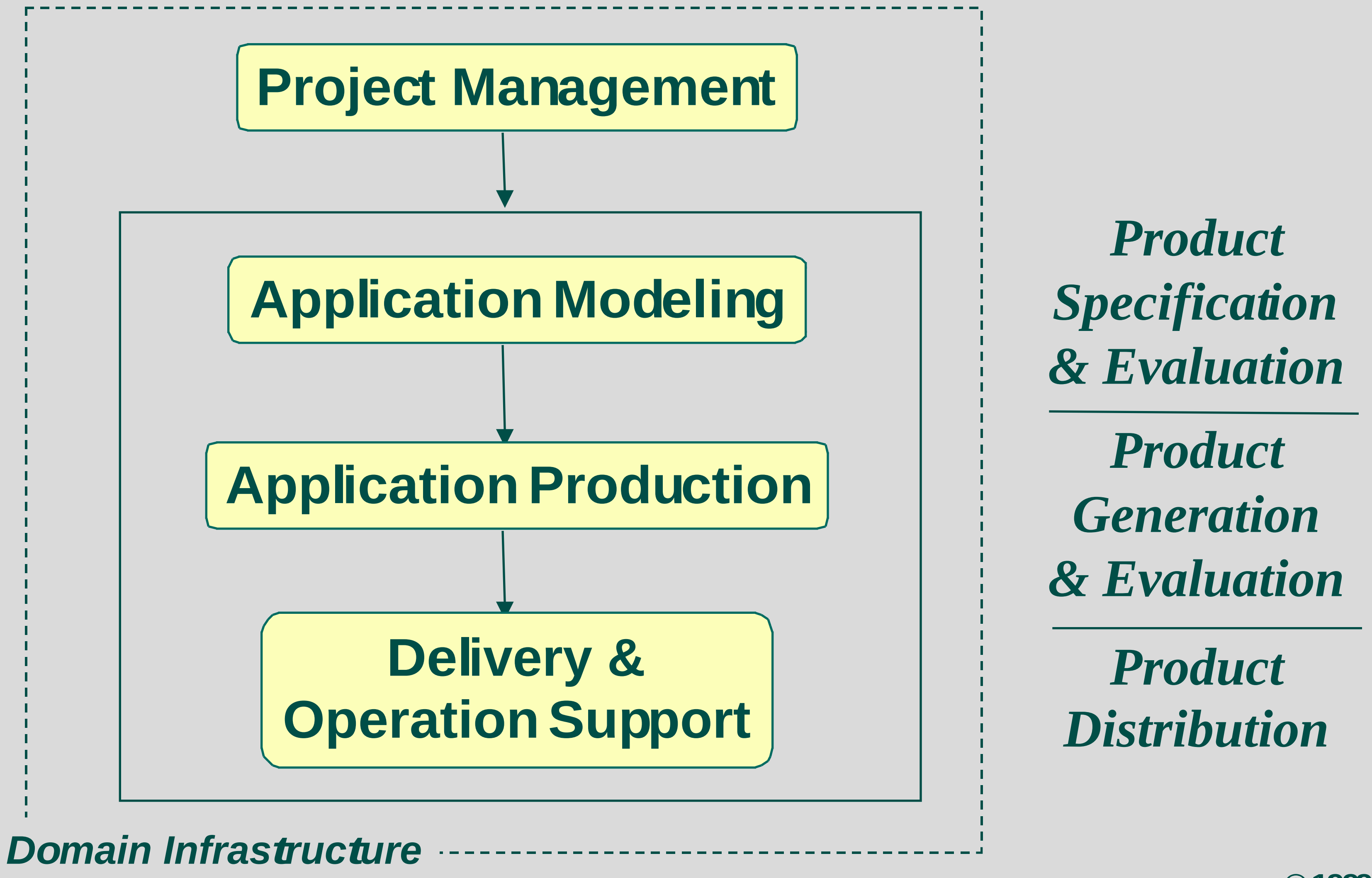
A Domain Engineering Process



A Conventional Application Engineering Process



A Streamlined Application Engineering Process



Benefits of DsE

- Customer needs expressed in a standardized, abbreviated form and terminology ensures clearer communication and earlier discovery of unsupported needs.
- Quality improvements in the product family improve the quality of all products.
- Process standardization fosters more predictable schedules and cost estimates.
- Process streamlining, based on a product family, reduces time and effort to deliver similar products.
- Problem and solution knowledge and expertise are more easily shared and extended.

For Additional Information on DsE and Adaptable Components

Prosperity Heights Software

www.domain-specific.com

info@domain-specific.com

1 703 573 3139

GradyCampbell@acm.org