Software Reuse

Week 9: More Software Implementation Issues

Edmund Yu, PhD
Associate Professor
esyu@syr.edu

SYRACUSE UNIVERSITY
ENGINEERING & COMPUTER SCIENCE

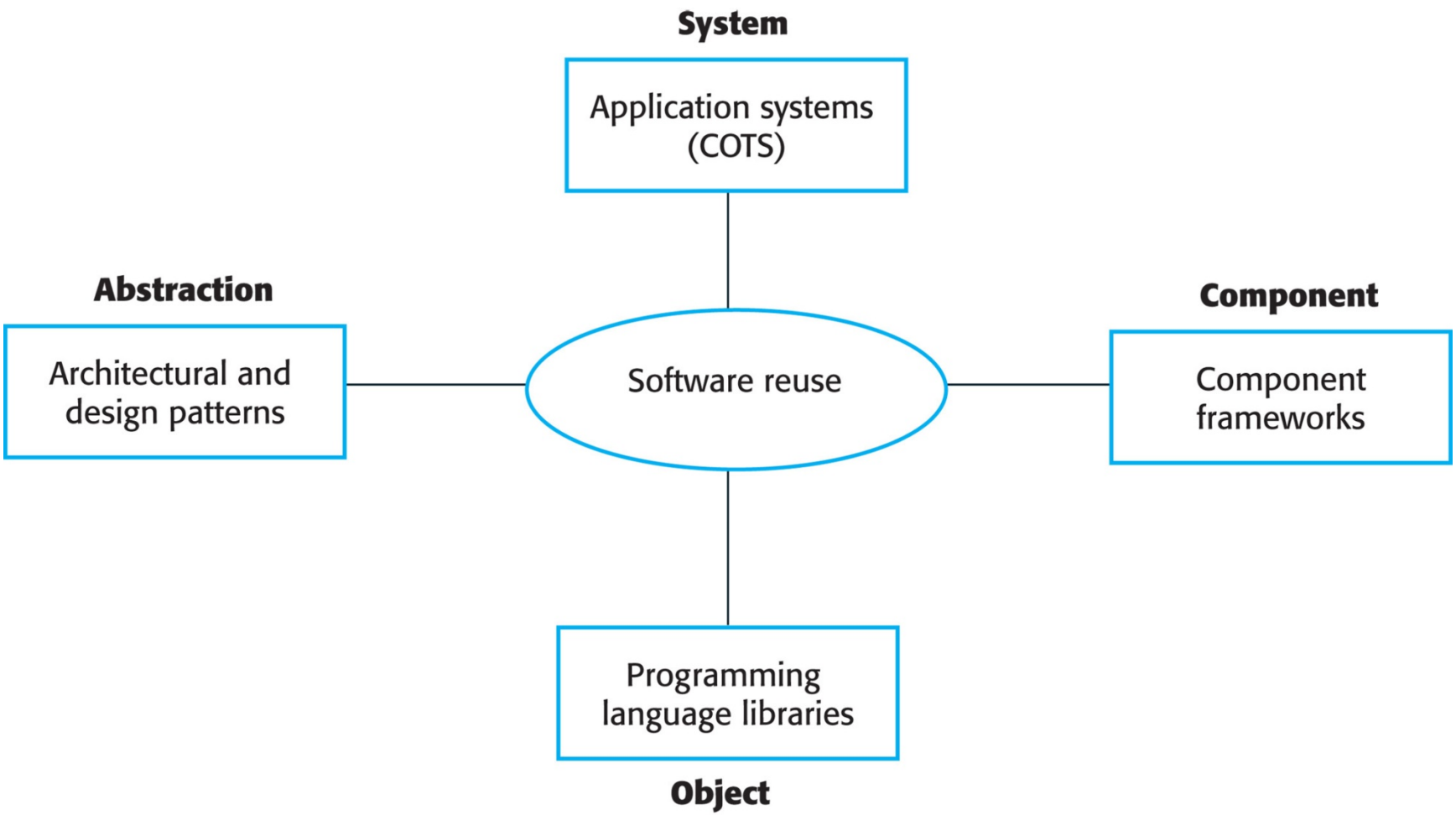
Software Reuse

- In most engineering disciplines, systems are designed by composing existing components that have been used in other systems.
- From the 1960s to the 1990s, most new software was developed from scratch.
  - The only significant reuse or software was the reuse of functions and objects in programming language libraries.
- Costs and schedule pressure mean that this approach became increasingly unviable, especially for commercial and Internet-based systems.
- An approach to development based around the reuse of existing software emerged and is now generally used for business and scientific software.

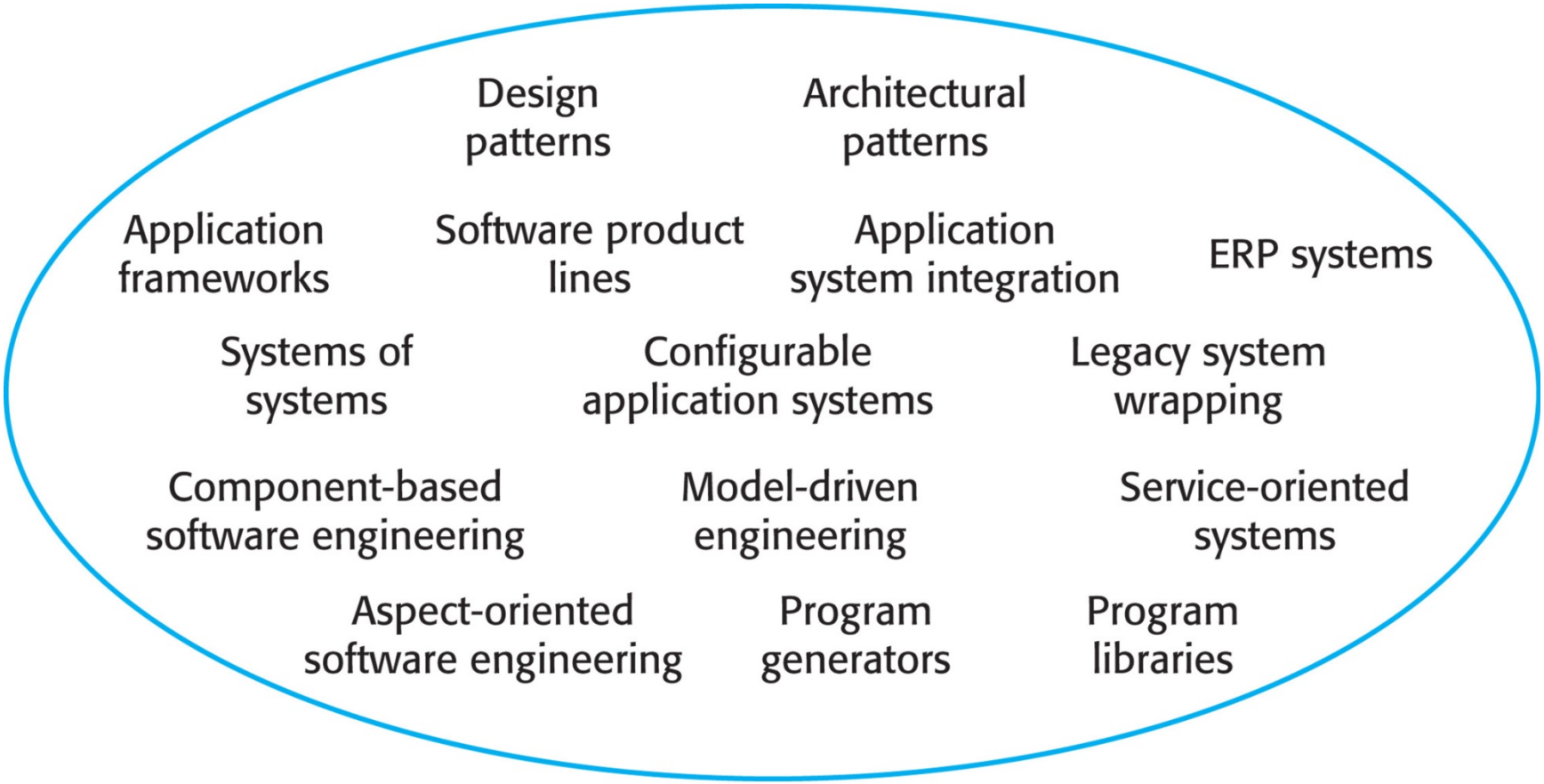
Reuse Levels

- The abstraction level
  - At this level, you don’t reuse software directly but use knowledge of successful abstractions in the design of your software.
- The object level
  - At this level, you directly reuse objects from a library rather than writing the code yourself.
- The component level
  - Components are collections of objects and object classes that you reuse in application systems.
- The system level
  - At this level, you reuse entire application systems.

Reuse Levels



The Reuse Landscape



Reuse Costs

- The costs of the time spent in looking for software to reuse and assessing whether or not it meets your needs.
- Where applicable, the costs of buying the reusable software. For large off-the-shelf systems, these costs can be very high.
- The costs of adapting and configuring the reusable software components or systems to reflect the requirements of the system that you are developing.
- The costs of integrating reusable software elements with each other (if you are using software from different sources) and with the new code that you have developed.

SYRACUSE UNIVERSITY
ENGINEERING & COMPUTER SCIENCE

Application Frameworks

Week 9: More Software Implementation Issues

Edmund Yu, PhD
Associate Professor
esyu@syr.edu

SYRACUSE UNIVERSITY
ENGINEERING & COMPUTER SCIENCE

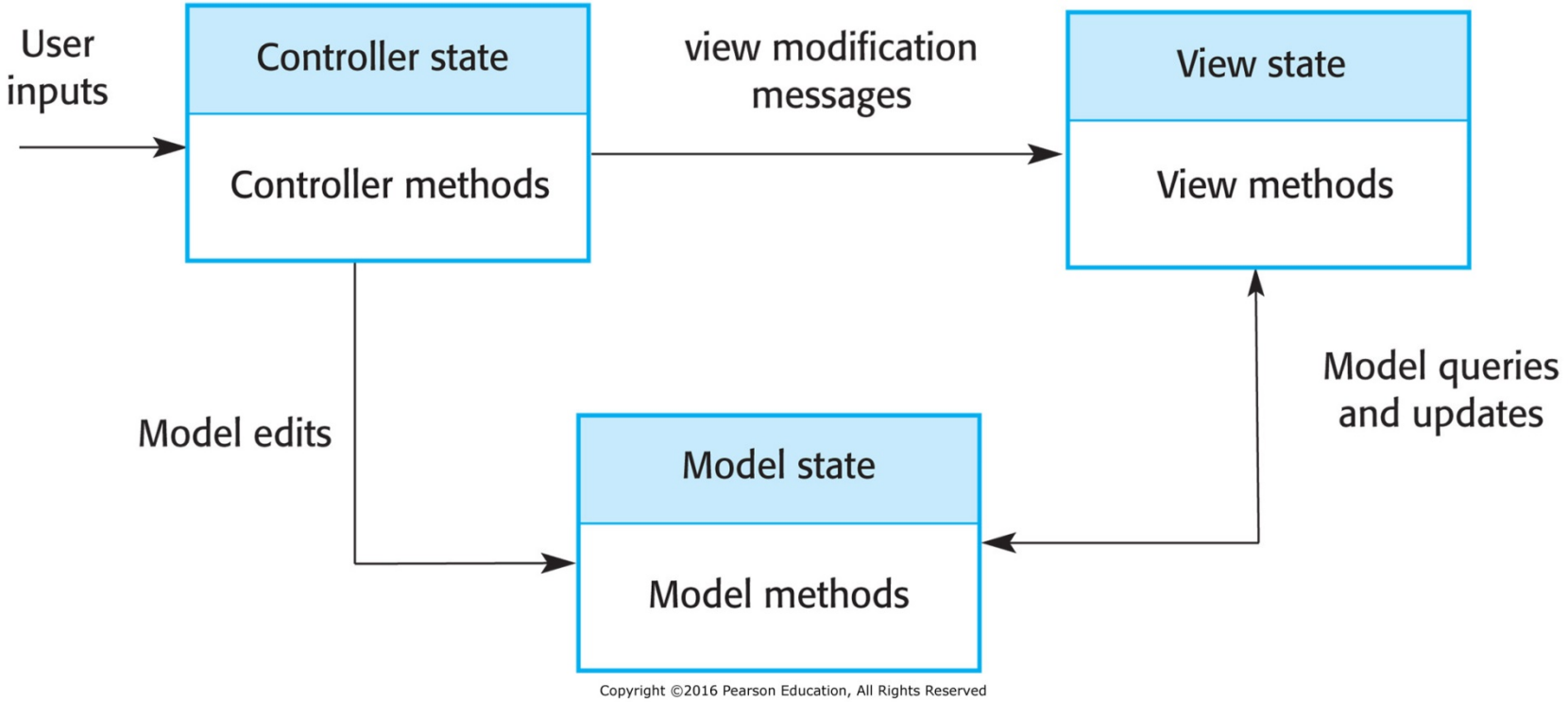
Application Frameworks

- Frameworks are moderately large entities that can be reused. They are somewhere between system and component reuse.
- Frameworks are a subsystem design made up of a collection of abstract and concrete classes and the interfaces between them.
- The subsystem is implemented by adding components to fill in parts of the design and by instantiating the abstract classes in the framework.

Web Application Frameworks

- Support the construction of dynamic websites as a front end for web applications.
- WAFs are now available for all of the commonly used web programming languages, e.g., Java, Python, Ruby, and so on.
- Interaction model is based on the model-view-controller (MVC) pattern.

Web Application Frameworks



- System infrastructure framework for GUI design.
- Allows for multiple presentations of an object and separate interactions with these presentations.
- MVC framework involves the instantiation of a number of patterns.

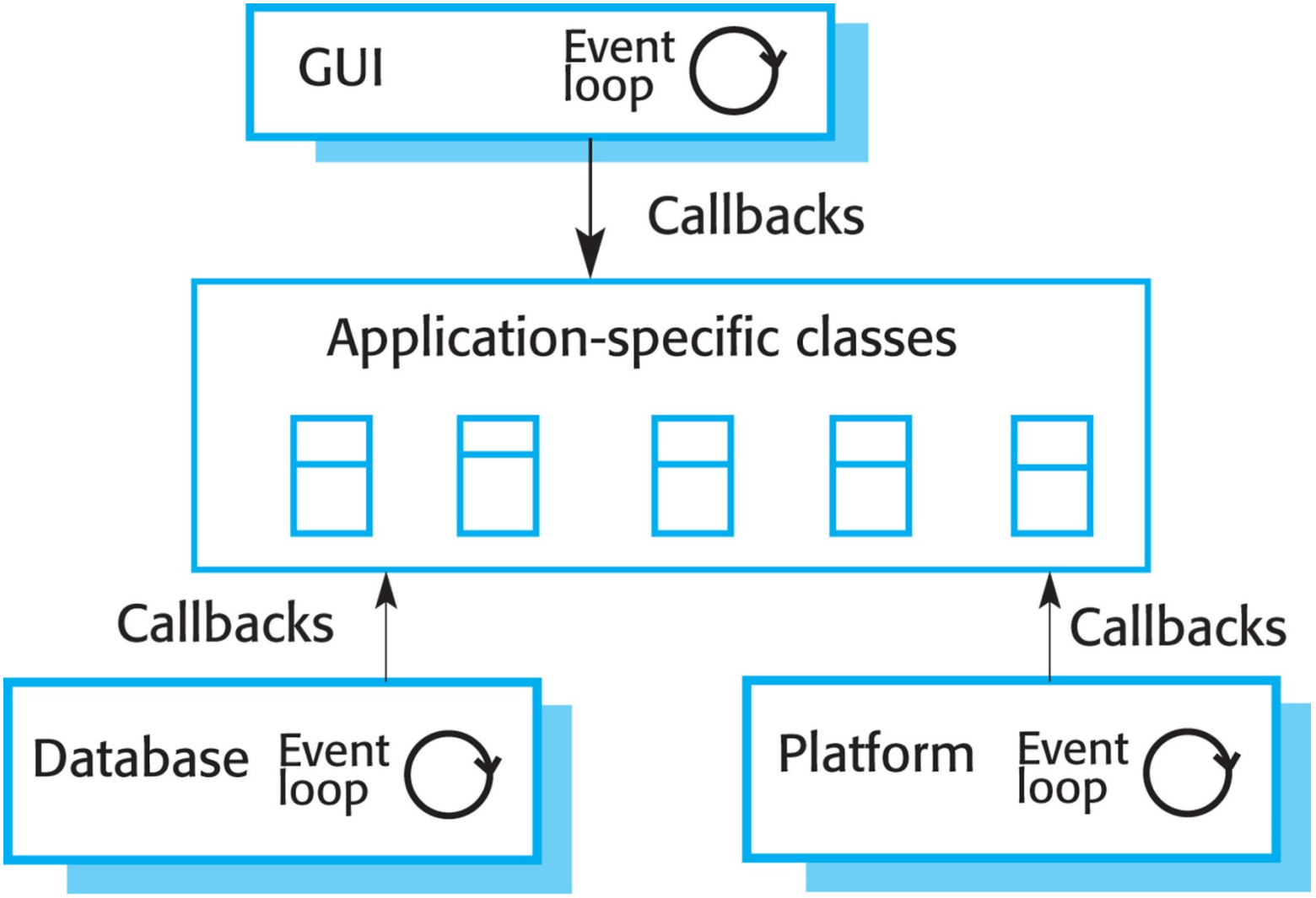
WAF Features

- **Security**—WAFs may include classes to help implement user authentication (login) and access.
- **Dynamic web pages**—Classes are provided to help you define web page templates and to populate these dynamically from the system database.
- **Database support**—The framework may provide classes that provide an abstract interface to different databases.
- **Session management**—Classes to create and manage sessions (a number of interactions with the system by a user) are usually part of a WAF.
- **User interaction**—Most web frameworks now provide AJAX support, which allows more interactive web pages to be created.

Extending Frameworks

- Frameworks are generic and are extended to create a more specific application or subsystem. They provide a skeleton architecture for the system.
- Extending the framework involves
  - Adding concrete classes that inherit operations from abstract classes in the framework
  - Adding methods that are called in response to events that are recognized by the framework
- Problem with frameworks is their complexity, which means that it takes a long time to use them effectively.

Inversion of Control in Frameworks



Framework Classes

- System infrastructure frameworks
  - Support the development of system infrastructures such as communications, user interfaces, and compilers
- Middleware integration frameworks
  - Standards and classes that support component communication and information exchange
- Enterprise application frameworks
  - Support the development of specific types of application such as telecommunications or financial systems

SYRACUSE UNIVERSITY
ENGINEERING & COMPUTER SCIENCE

Software Product Lines

Week 9: More Software Implementation Issues

Edmund Yu, PhD
Associate Professor
esyu@syr.edu

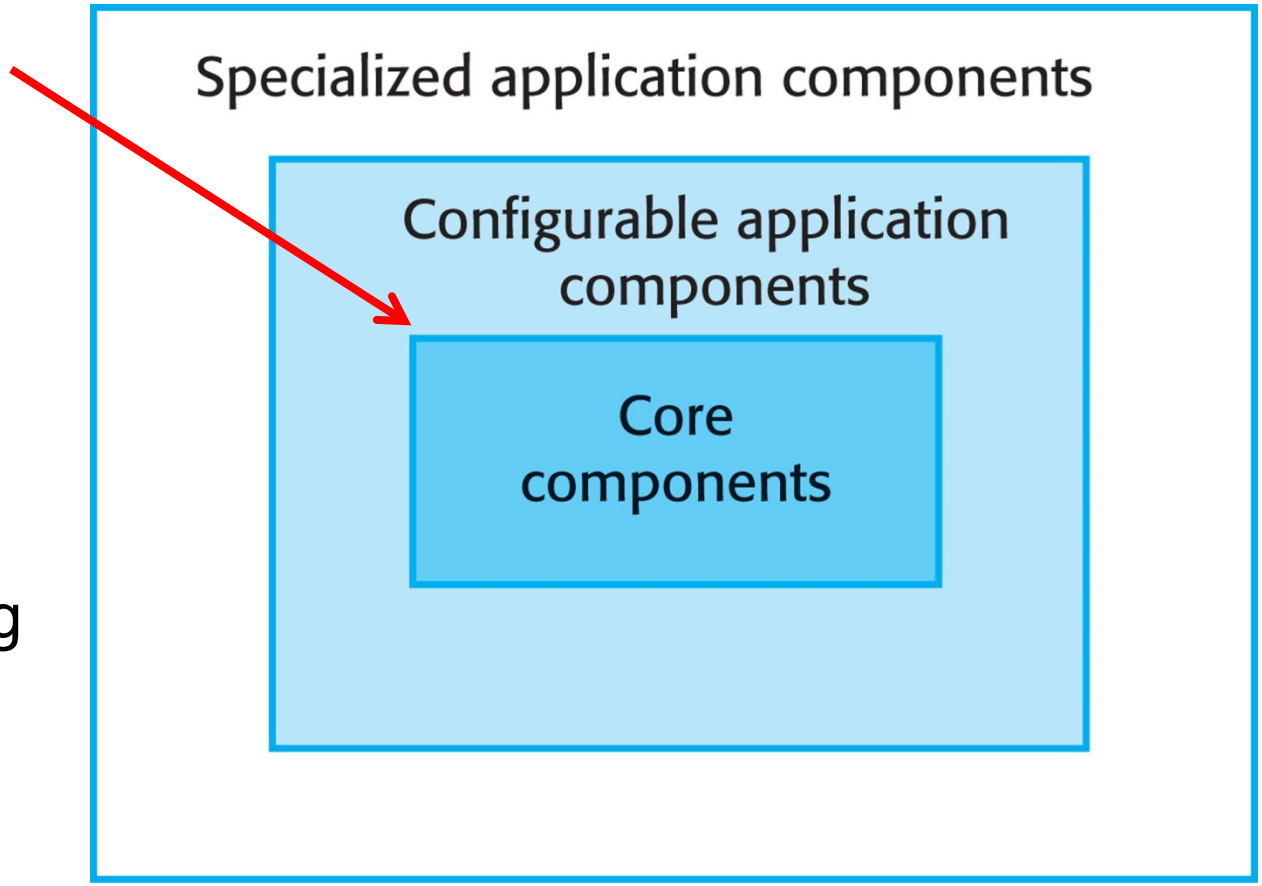
SYRACUSE UNIVERSITY
ENGINEERING & COMPUTER SCIENCE

Software Product Lines

- Software product lines or application families are applications with generic functionality that can be adapted and configured for use in a specific context.
- A software product line is a set of applications with a common architecture and shared components, with each application specialized to reflect different requirements.
- Adaptation may involve:
  - Component and system configuration
  - Adding new components to the system
  - Selecting from a library of existing components
  - Modifying components to meet new requirements

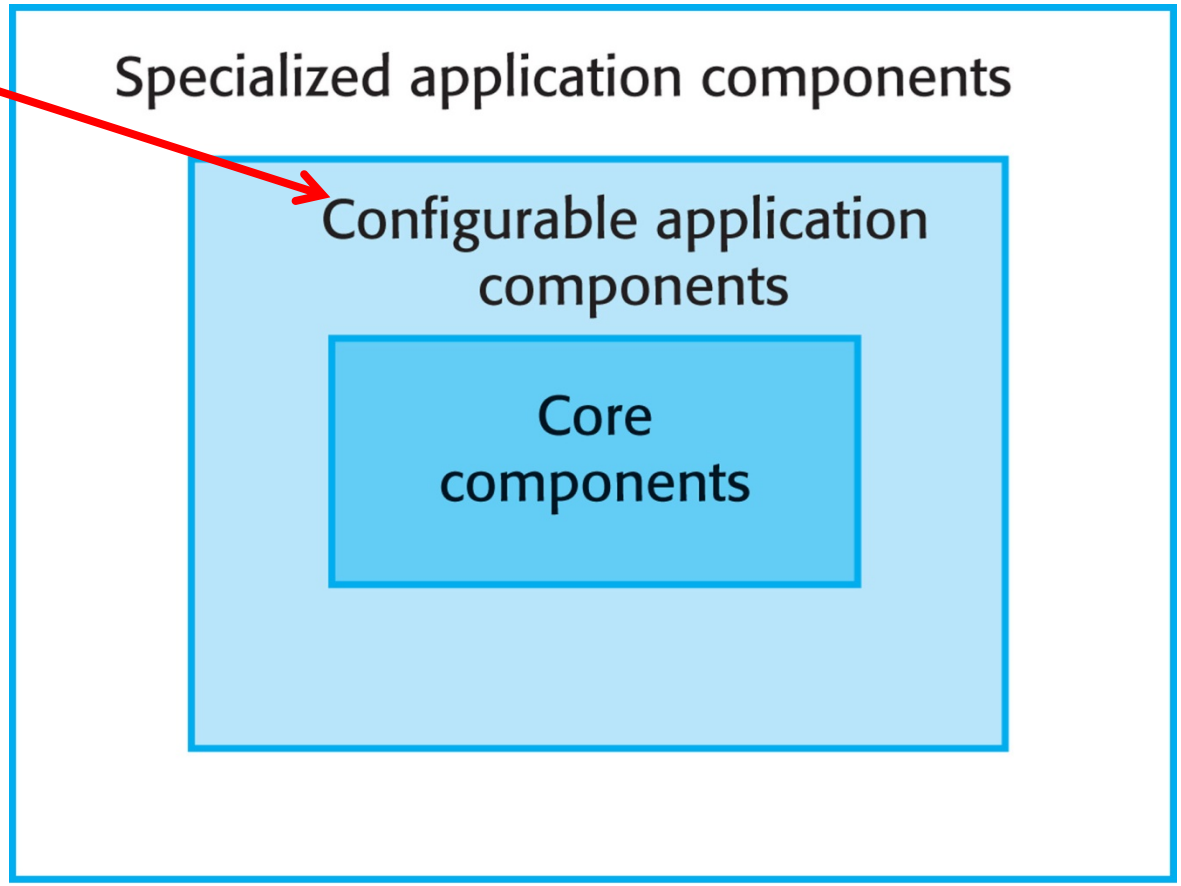
Base Systems for a Software Product Line

- Core components provide infrastructure support.
- These are not usually modified when developing a new instance of the product line.



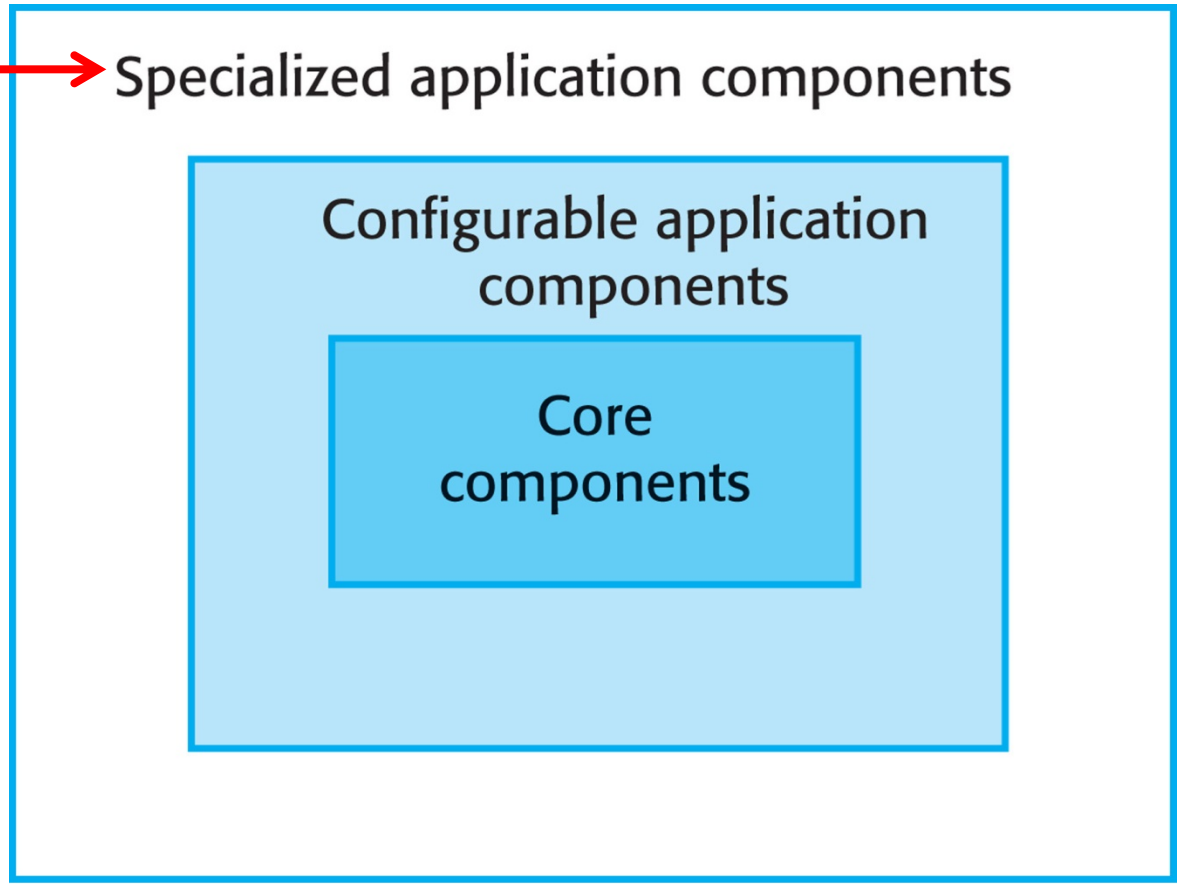
Base Systems for a Software Product Line

- Configurable components may be modified and configured to specialize them for a new application.
- It is possible to reconfigure these components without changing their code.



Base Systems for a Software Product Line

- Specialized, domain-specific components may be replaced when a new instance of a product line is created.



Application Frameworks vs. Product Lines

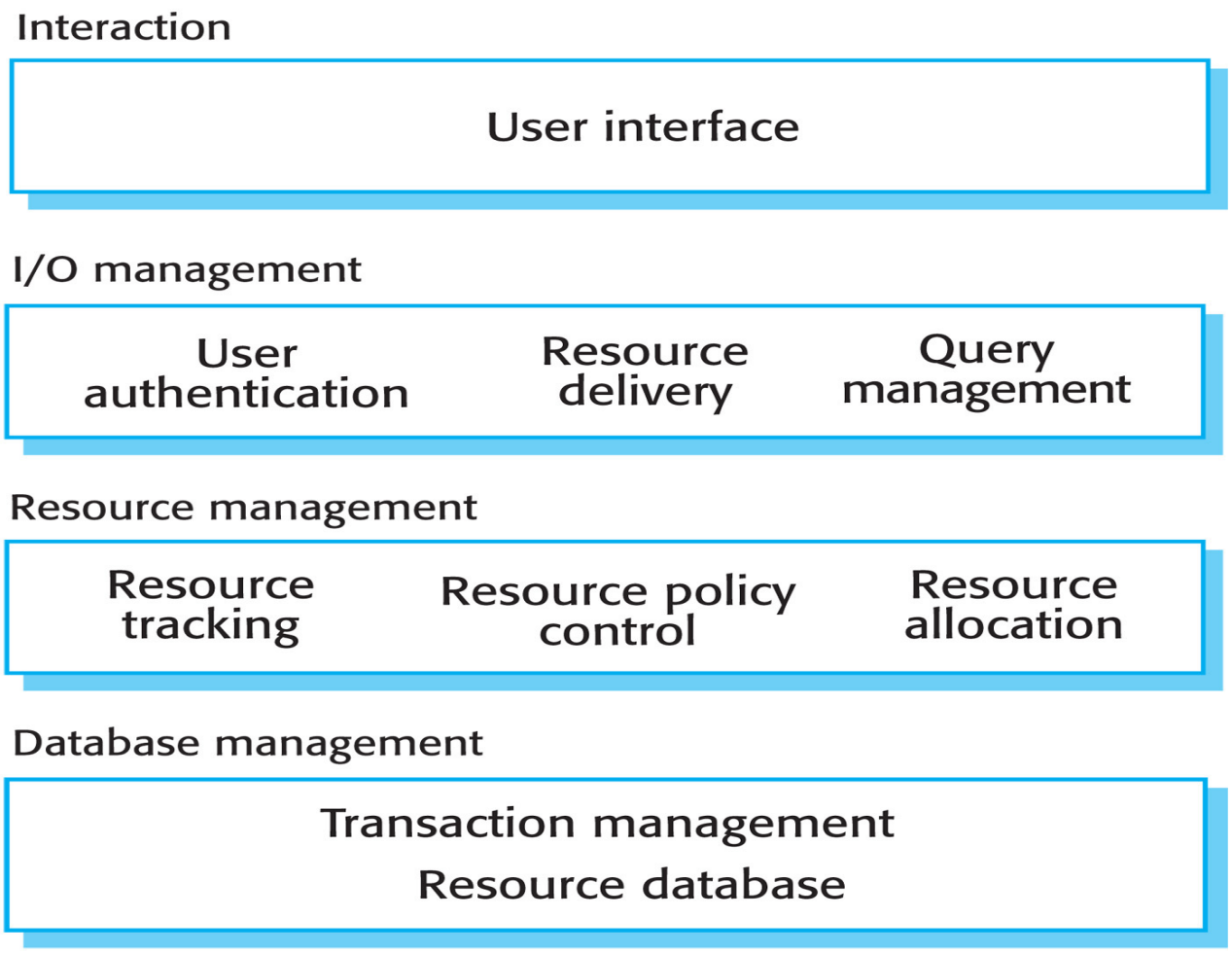
- Application frameworks rely on object-oriented features such as polymorphism to implement extensions while software product lines need not be object-oriented.
- Application frameworks focus on providing technical rather than domain-specific support, while software product lines embed domain and platform information.
- Product lines often control applications for equipment.
- Software product lines are made up of a family of applications usually owned by the same organization.

Product Line Architectures

- Architectures must be structured in such a way to separate different subsystems and to allow them to be modified.
- The architecture should also separate entities and their descriptions, and the higher levels in the system access entities through descriptions rather than directly.

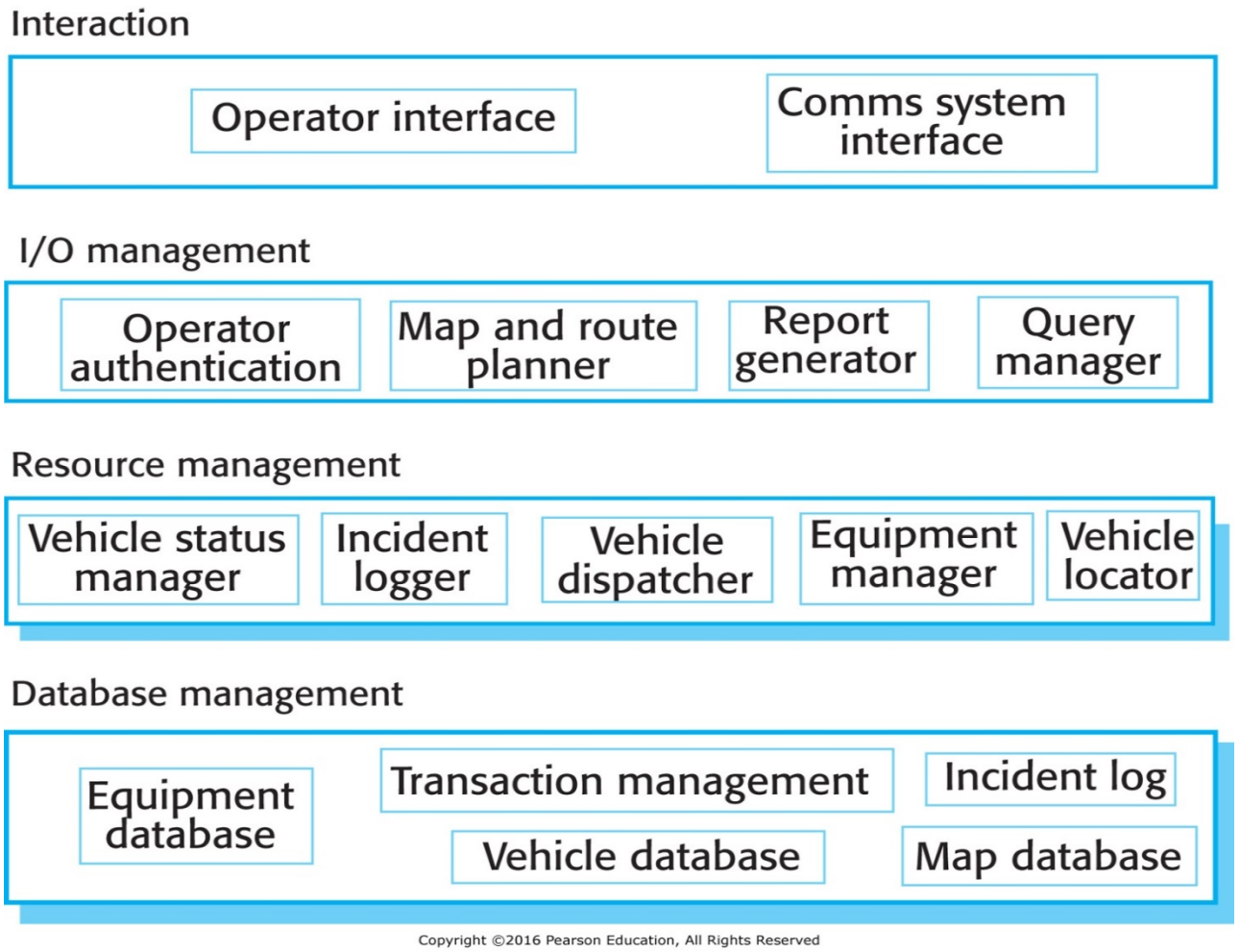
Product Line Architectures: An Example

The architecture of a resource management system



Product Line Architectures: An Example

A product-line architecture of a vehicle dispatcher system

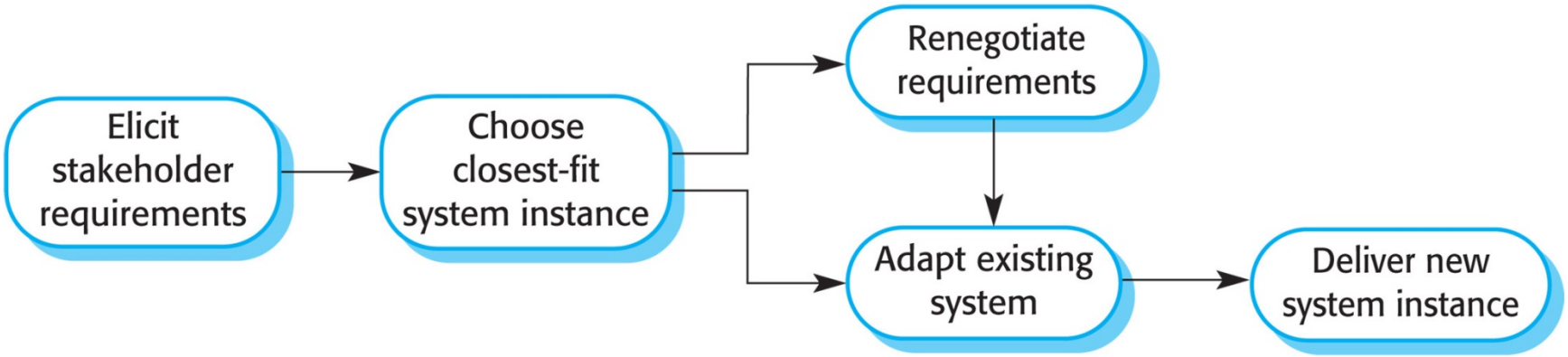


Product Line Specialization

- Platform specialization
  - Different versions of the application are developed for different platforms.
- Environment specialization
  - Different versions of the application are created to handle different operating environments (e.g., different types of communication equipment).
- Functional specialization
  - Different versions of the application are created for customers with different requirements.
- Process specialization
  - Different versions of the application are created to support different business processes.

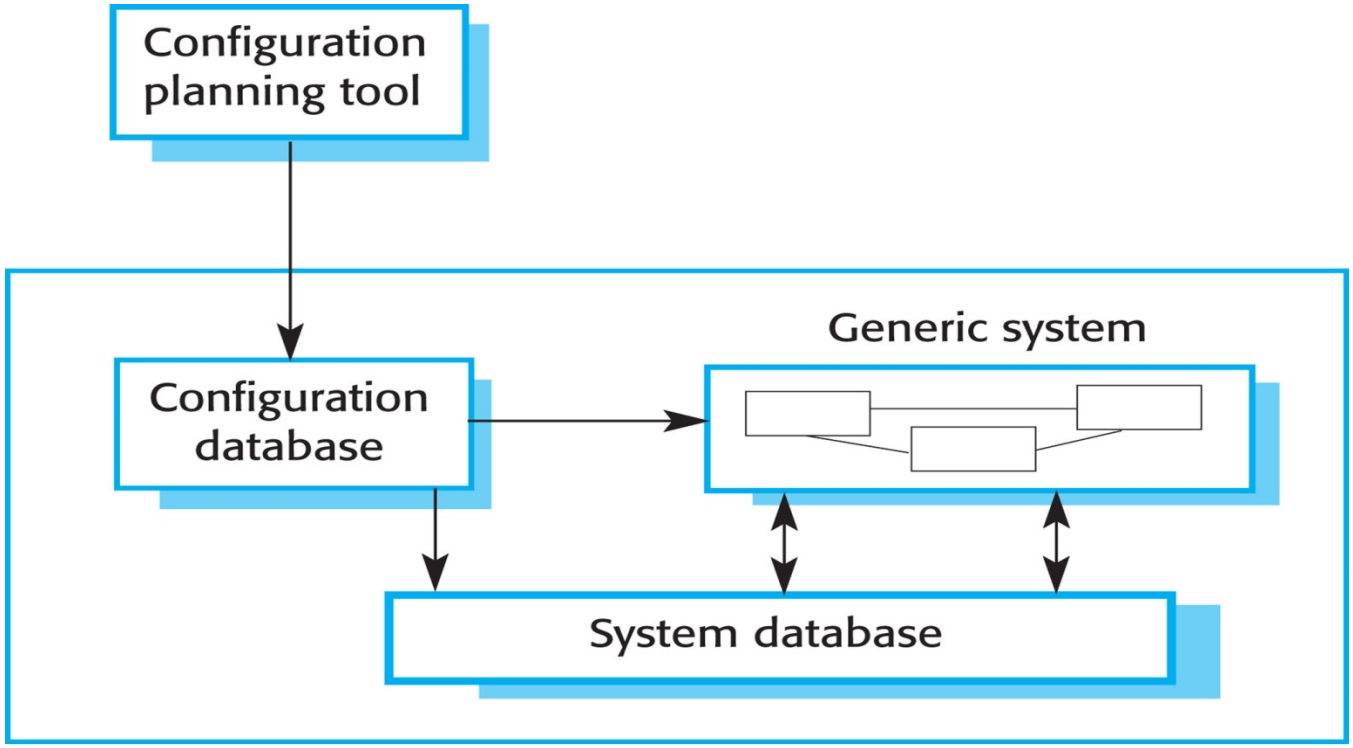
Product Instance Development

1. Elicit stakeholder requirements: Use existing family member as a prototype.
2. Choose closest-fit family member: Find the family member that best meets the requirements.
3. Renegotiate requirements: Adapt requirements as necessary to capabilities of the software.
4. Adapt existing system: Develop new modules and make changes for family member.
5. Deliver new family member: Document key features for further member development.



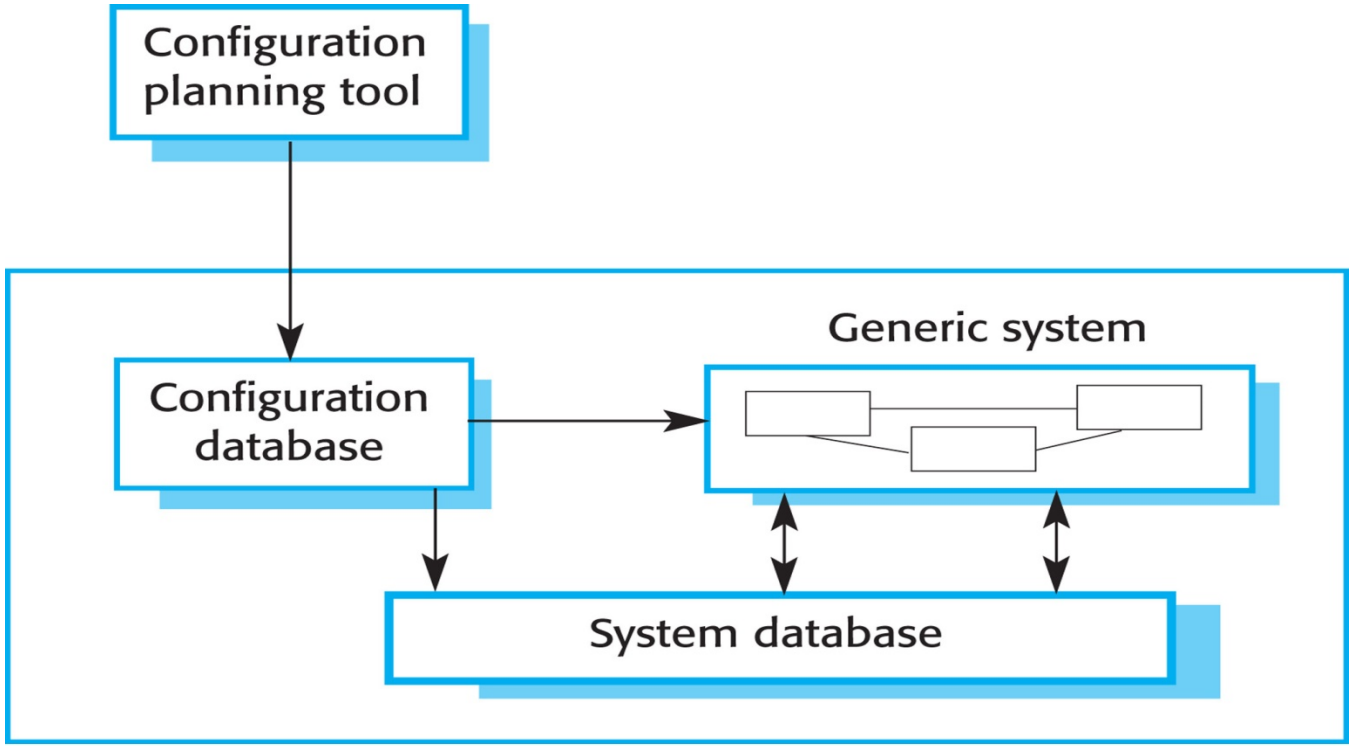
Product Line Configuration

- Design-time configuration
  - The organization that is developing the software modifies a common product line core by developing, selecting, or adapting components to create a new system for a customer.



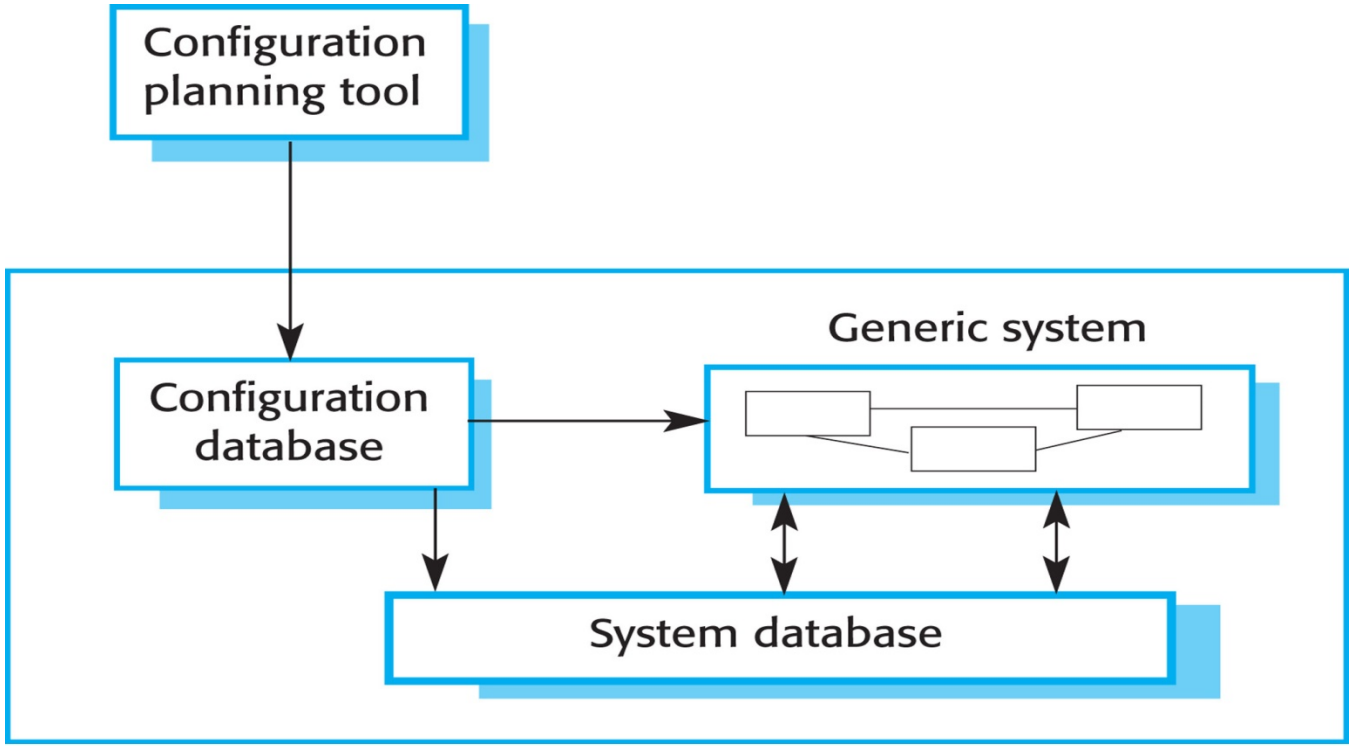
Product Line Configuration

- Deployment-time configuration
  - A generic system is designed for configuration by a customer or consultants working with the customer.



Product Line Configuration

- Deployment-time configuration
  - Knowledge of the customer's specific requirements and the system's operating environment is embedded in configuration data that are used by the generic system.



SYRACUSE UNIVERSITY
ENGINEERING & COMPUTER SCIENCE

Application System Reuse

Week 9: More Software Implementation Issues

Edmund Yu, PhD
Associate Professor
esyu@syr.edu

SYRACUSE UNIVERSITY
ENGINEERING & COMPUTER SCIENCE

Application System Reuse

- An application system product is a software system that can be adapted for different customers without changing the source code of the system.
- Application systems have generic features and so can be used/reused in different environments.
- Application system products are adapted by using built-in configuration mechanisms that allow the functionality of the system to be tailored to specific customer needs.
  - For example, in a hospital patient record system, separate input forms and output reports might be defined for different types of patient.

Benefits of Application System Reuse

- As with other types of reuse, more rapid deployment of a reliable system may be possible.
- It is possible to see what functionality is provided by the applications and so it is easier to judge whether or not they are likely to be suitable.
- Some development risks are avoided by using existing software. However, this approach has its own risks.
- Businesses can focus on their core activity without having to devote a lot of resources to IT systems development.
- As operating platforms evolve, technology updates may be simplified as these are the responsibility of the COTS (commercial off-the-shelf) product vendor rather than the customer.

Problems of Application System Reuse

- Requirements usually have to be adapted to reflect the functionality and mode of operation of the COTS product.
- The COTS product may be based on assumptions that are practically impossible to change.
- Choosing the right COTS system for an enterprise can be a difficult process, especially as many COTS products are not well documented.
- There may be a lack of local expertise to support systems development.
- The COTS product vendor controls system support and evolution.

Configurable Application Systems

- Configurable application systems are generic application systems that may be designed to support a particular business type, business activity or, sometimes, a complete business enterprise.
  - For example, an application system may be produced for dentists that handles appointments, dental records, patient recall, and so on.
- Domain-specific systems, such as systems to support a business function (e.g., document management) provide functionality that is likely to be required by a range of potential users.

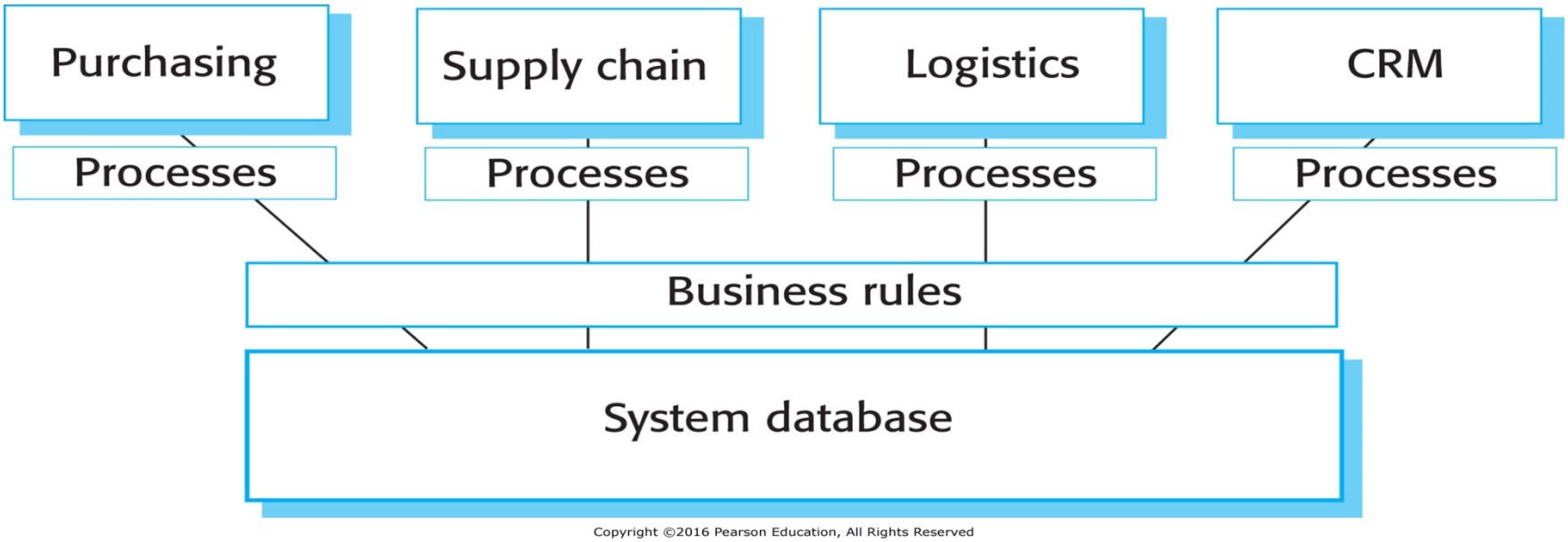
Configurable Application Systems

Configurable application systems	Application system integration
Single product that provides the functionality required by a customer	Several different application systems are integrated to provide customized functionality
Based on a generic solution and standardized processes	Flexible solutions may be developed for customer processes
Development focus is on system configuration	Development focus is on system integration
System vendor is responsible for maintenance	System owner is responsible for maintenance
System vendor provides the platform for the system	System owner provides the platform for the system

ERP Systems

- An enterprise resource planning (ERP) system is a generic system that supports common business processes such as ordering and invoicing, manufacturing, and so on.
- These are very widely used in large companies—they represent probably the most common form of software reuse.
- The generic core is adapted by including modules and by incorporating knowledge of business processes and rules.

The Architecture of an ERP System



- A number of modules to support different business functions
- A defined set of business processes, associated with each module, which relate to activities in that module
- A common database that maintains information about all related business functions
- A set of business rules that apply to all data in the database

ERP Configuration

- Selecting the required functionality from the system
- Establishing a data model that defines how the organization's data will be structured in the system database
- Defining business rules that apply to that data
- Defining the expected interactions with external systems
- Designing the input forms and the output reports generated by the system
- Designing new business processes that conform to the underlying process model supported by the system
- Setting parameters that define how the system is deployed on its underlying platform

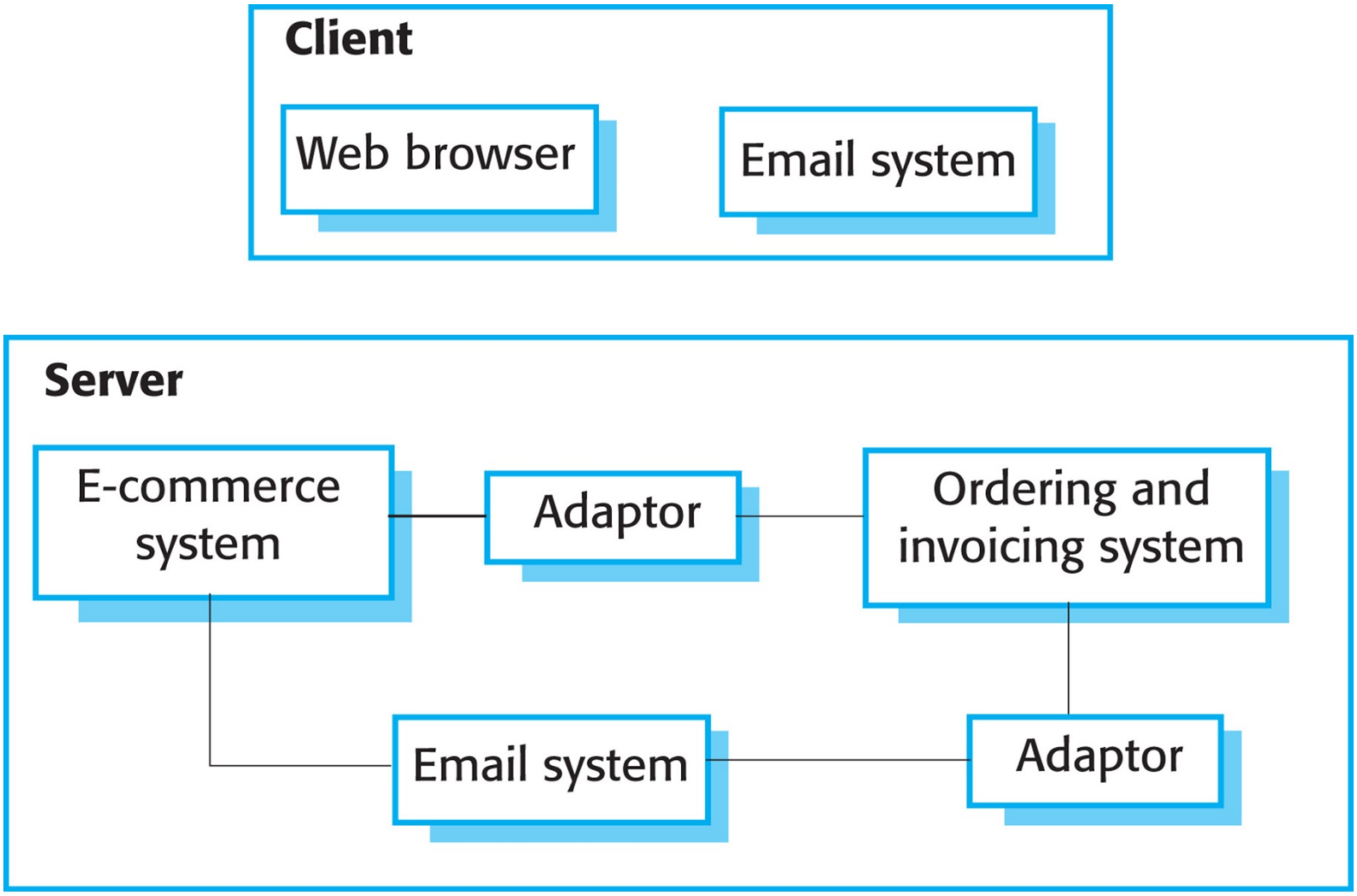
Integrated Application Systems

- Integrated application systems are applications that include two or more application system products and/or legacy application systems.
- You may use this approach when there is no single application system that meets all of your needs or when you wish to integrate a new application system with systems that you already use.

Design Choices

- Which individual application systems offer the most appropriate functionality?
  - Typically, there will be several application system products available, which can be combined in different ways.
- How will data be exchanged?
  - Different products normally use unique data structures and formats. You have to write adaptors that convert from one representation to another.
- What features of a product will actually be used?
  - Individual application systems may include more functionality than you need, and functionality may be duplicated across different products.

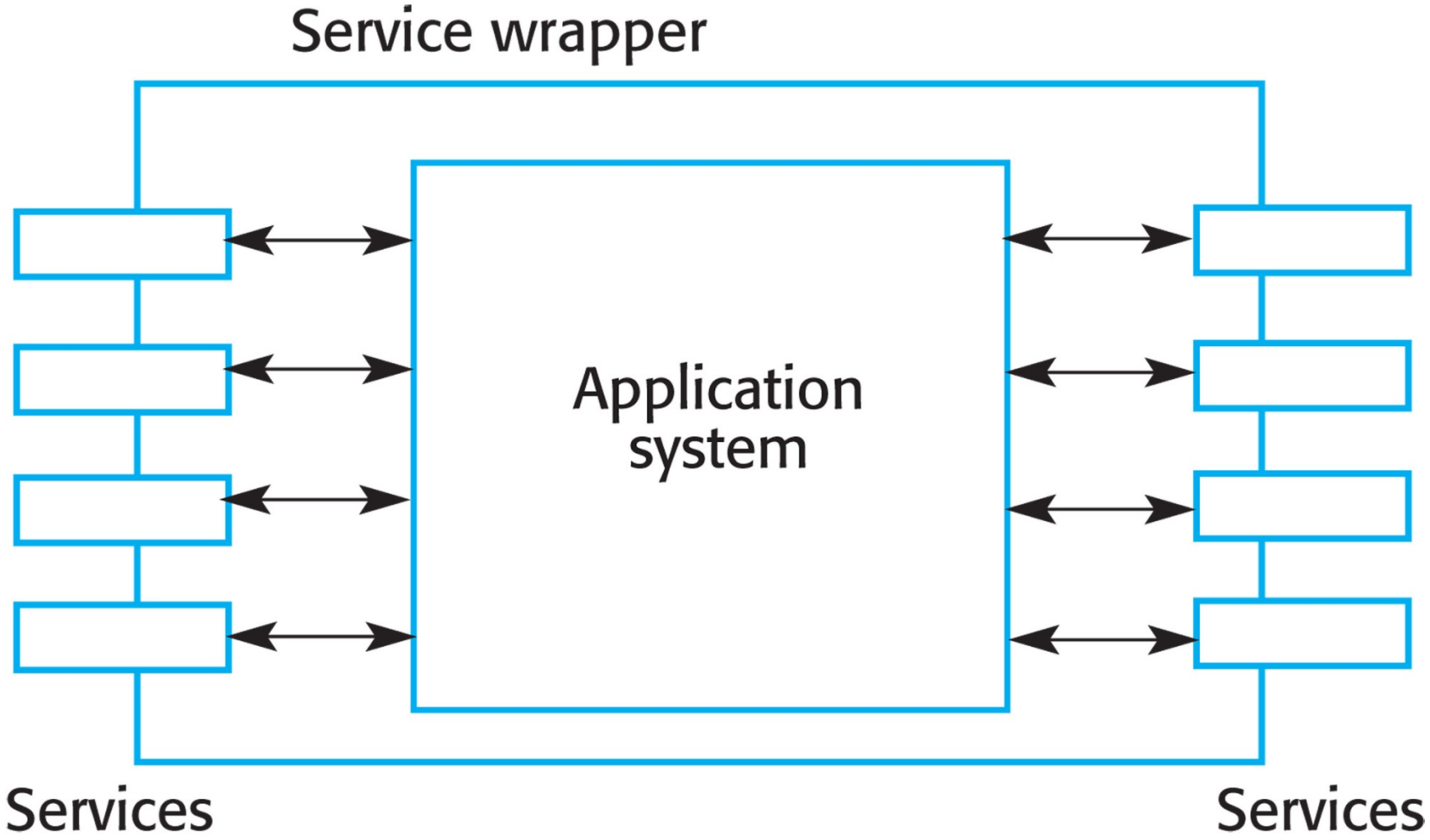
An Integrated Procurement System



Service-Oriented Interfaces

- Application system integration can be simplified if a service-oriented approach is used.
- A service-oriented approach means allowing access to the application system's functionality through a standard service interface, with a service for each discrete unit of functionality.
- Some applications may offer a service interface but, sometimes, this service interface has to be implemented by the system integrator. You have to program a wrapper that hides the application and provides externally visible services.

Application Wrapping



Application System Integration Problems

- Lack of control over functionality and performance
  - Application systems may be less effective than they appear.
- Problems with application system interoperability
  - Different application systems may make different assumptions that means integration is difficult.
- No control over system evolution
  - Application system vendors, not system users, control evolution.
- Support from system vendors
  - Application system vendors may not offer support over the lifetime of the product.

SYRACUSE UNIVERSITY
ENGINEERING & COMPUTER SCIENCE

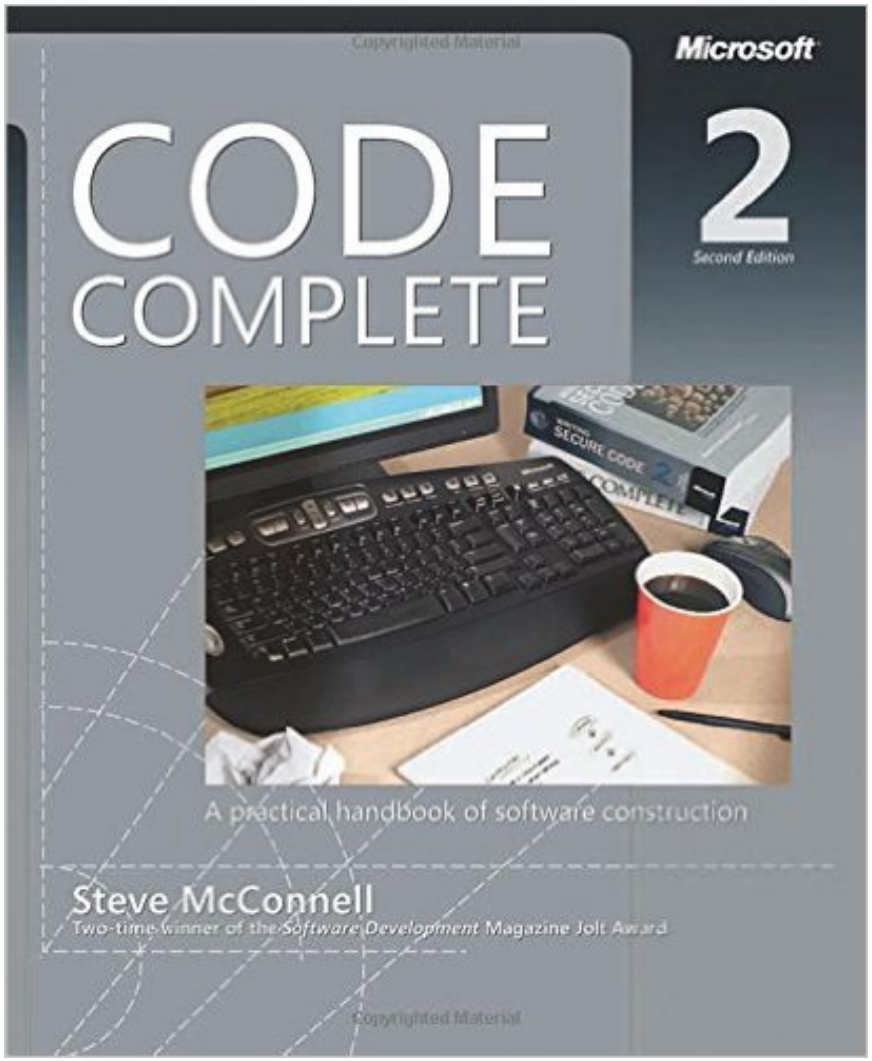
Creating High-Quality Class Interfaces

Week 9: More Software Implementation Issues

Edmund Yu, PhD
Associate Professor
esyu@syr.edu

SYRACUSE UNIVERSITY
ENGINEERING & COMPUTER SCIENCE

Code Complete



Classes

- Today, programmers think about programming in terms of classes.
  - A class is a collection of data and routines that share a **cohesive**, **well-defined** responsibility.
    - The single responsibility principle
- A key to being an effective programmer is maximizing the portion of a program that you can safely ignore.
  - Classes are the primary tool for accomplishing that objective.

Foundations of Classes: ADT

- An abstract data type (**ADT**) is a collection of data and operations that works on that data.
  - The operations both describe the data to the rest of the program and allow the rest of the program to change the data.
- Understanding ADTs is essential to understanding object-oriented programming.
  - Without understanding ADTs, programmers create classes that are “classes” in name only.
    - In reality, they are little more than convenient carrying cases for loosely related collections of data and routines.
  - With an understanding of ADTs, programmers can create classes that are easier to implement initially and easier to modify over time.

ADTs: An Example

- Suppose you're writing software that controls the cooling system for a nuclear reactor.
  - You can treat the cooling system as an abstract data type by defining the following operations for it:

```
coolingSystem.GetTemperature()
coolingSystem.SetCirculationRate(rate)
coolingSystem.OpenValve(valveNumber)
coolingSystem.CloseValve(valveNumber)
```

ADTs: More Examples

Stack

Initialize stack

Push item onto stack

Pop item from stack

Read top of stack

List

Initialize list

Insert item in list

Remove item from list

Read next item from list

Menu

Start new menu

Delete menu

Add menu item

Remove menu item

Activate menu item

Deactivate menu item

Display menu

Hide menu

Get menu choice

File

Open file

Read file

Write file

Set current file location

Close file

Elevator

Move up one floor

Move down one floor

Move to specific floor

Report current floor

Return to home floor

ADTs and Classes

- Abstract data types form the foundation for the concept of classes.
  - In languages that support classes, you can implement each abstract data type as its own class.
- Classes usually involve the additional concepts of **inheritance** and **polymorphism**.
  - One way of thinking of a class is as an abstract data type plus inheritance and polymorphism.

Good Class Interfaces

- The first and most important step in creating a high-quality class is creating a good class interface.

- This consists of creating a good **abstraction** for the interface to represent and ensuring that the details remain hidden behind the abstraction.

Good Abstraction

```
class Employee {
public:
    // public constructors & destructors
    Employee();
    Employee(
        FullName name,
        String address,
        String workPhone,
        String homePhone,
        TaxId taxIdNumber,
        JobClassification jobClass
    );
    virtual ~Employee();
    // public routines
    FullName GetName() const;
    String GetAddress() const;
    String GetWorkPhone() const;
    String GetHomePhone() const;
    TaxId GetTaxIdNumber() const;
    JobClassification GetJobClassification() const;
    ...
private:
    ...
};
```

Good vs. Poor Abstractions

```
class Program {
   public:
   // public routines
   void InitializeCommandStack();
   void PushCommand(Command
&command );
   Command PopCommand();
   void ShutdownCommandStack();
   void InitializeReportFormatting();
   void FormatReport(Report &report );
   void PrintReport(Report &report );
   void InitializeGlobalData();
   void ShutdownGlobalData();
   ...
   private:
   ...
}
```

```
class Program {
   public:
   // public routines
   void InitializeProgram();
   void ShutDownProgram();
   ...
   private:
   ...
}
```

Poor Abstractions

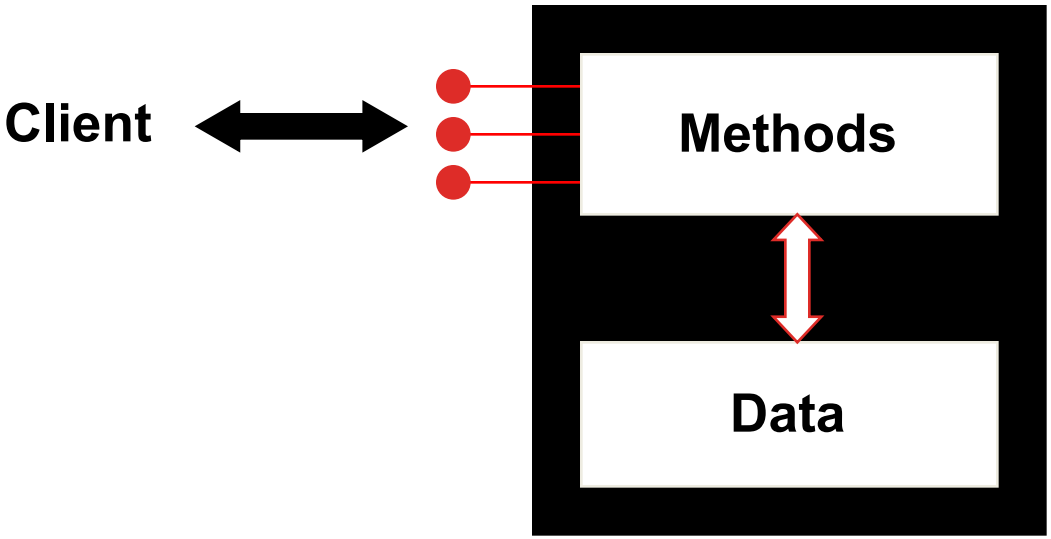
```
class EmployeeList: public ListContainer {
   public:
   ...
   // public routines
   void AddEmployee(Employee &employee );
   void RemoveEmployee(Employee &employee );

   Employee NextItemInList( Employee &employee );
   Employee FirstItem( Employee &employee );
   Employee LastItem( Employee &employee );
   ...
   private:
   ...
}
```

Better Abstractions

```
class EmployeeList {
   public:
   ...
   // public routines
   void AddEmployee( Employee &employee );
   void RemoveEmployee( Employee &employee );
   Employee NextEmployee( Employee &employee );
   Employee FirstEmployee( Employee &employee );
   Employee LastEmployee( Employee &employee );
   ...
   private:
   ListContainer m_EmployeeList;
   ...
}
```

Abstraction vs. Encapsulation

- **Abstraction**
  - Is used to manage complexity by providing models that allow you to ignore implementation details
- **Encapsulation**
  - Is a stronger concept than abstraction.
  - It prevents you from looking at the details even if you want to. (The black-box view of an object.)



Encapsulation

- **Rules regarding encapsulation:**
  - Minimize accessibility of classes and members.
    - If you're wondering whether a specific routine should be public, private, or protected, one school of thought is that you should favor the strictest level of privacy that's workable.
  - Don't expose member data in public.
    - Exposing member data is a violation of encapsulation and limits your control over the abstraction.

Encapsulation

	Public	Private
Variables	Violate encapsulation	Enforce encapsulation
Methods	Provide services to clients	Support other methods in the class

Abstraction vs. Encapsulation

- A **Point** class that exposes:

```
float x;
float y;
float z;
```

  violates encapsulation.

- A **Point** class that exposes:

```
float GetX();
float GetY();
float GetZ();
void SetX( float x );
void SetY( float y );
void SetZ( float z );
```

  maintains perfect encapsulation.

SYRACUSE UNIVERSITY
ENGINEERING & COMPUTER SCIENCE

Creating High-Quality Routines

Week 9: More Software Implementation Issues

Edmund Yu, PhD
Associate Professor
esyu@syr.edu

SYRACUSE UNIVERSITY
ENGINEERING & COMPUTER SCIENCE

High-Quality Routines

- What is a high-quality routine?
  - This is a harder question. Perhaps the easiest answer is to show what a high-quality routine is not. (See the example on the next slide.)

A Poor-Quality Routine

```
void HandleStuff( CORP_DATA &inputRec, int crntQtr, EMP_DATA
    empRec, double &estimRevenue, double ytdRevenue, int screenX,
    int screenY, COLOR_TYPE &newColor, COLOR_TYPE &prevColor,
    StatusType &status, int expenseType )
{
int i;
for ( i = 0; i < 100; i++ ) {
inputRec.revenue[i] = 0;
inputRec.expense[i] = corpExpense[ crntQtr ][ i ];
}
UpdateCorpDatabase( empRec );
estimRevenue = ytdRevenue * 4.0 / (double) crntQtr;
newColor = prevColor;
status = SUCCESS;
```

A Poor-Quality Routine (cont.)

```
if ( expenseType == 1 ) {
for ( i = 0; i < 12; i++ )
profit[i] = revenue[i] - expense.type1[i];
}
else if ( expenseType == 2 ) {
profit[i] = revenue[i] - expense.type2[i];
}
else if ( expenseType == 3 )
profit[i] = revenue[i] - expense.type3[i];
}
```

High-Quality Routines

- **Cohesion** is a workhorse concept for the design of routines.
  - For routines, cohesion refers to how closely the operations in a routine are related.
  - Some programmers prefer the term "**strength**": How strongly related are the operations in a routine?
    - A function like ***Cosine()*** is perfectly cohesive because the whole routine is dedicated to performing one function.
    - A function ***CosineAndTan()*** has lower cohesion because it tries to do more than one thing.
  - The goal is to have each routine do one thing well and not do anything else (single responsibility).

High-Quality Routines

- The payoff is higher **reliability**.
  - One study of 450 routines found that **50** percent of the highly cohesive routines were fault free, whereas only **18** percent of routines with low cohesion were fault free (Card, Church, and Agresti 1986).

Levels of Cohesion

- Discussions about cohesion typically refer to the following levels of cohesion:
  - **Functional cohesion** (the strongest and best kind of cohesion) occurs when a routine performs one and only one operation.
    - E.g., *sin()*, *GetCustomerName()*, *EraseFile()*, *CalculateLoanPayment()*, *AgeFromBirthday()*…
  - **Sequential cohesion** occurs when a routine contains operations that must be performed in a specific order, that share data from step to step, and that don't make up a complete function when done together.
    - E.g., a routine that calculates an employee's age and time to retirement, given a birth date

Levels of Cohesion

- **Communicational cohesion** occurs when operations in a routine make use of the same data but aren't related in any other way.
  - If a routine prints a summary report and then reinitializes/updates the summary data, the routine has communicational cohesion.
- **Temporal cohesion** occurs when operations are combined into a routine because they are all done at the same time.
  - E.g., *Startup()*, *Shutdown()*…
  - Some programmers consider temporal cohesion to be unacceptable because it's sometimes associated with bad programming.

Levels of Cohesion

- **Procedural cohesion** occurs when operations in a routine are done in a specified order.
  - E.g., a routine that gets an employee name, then an address, and then a phone number.
    - The order of these operations is important only because it matches the order in which the user is asked for the data on the input screen.
  - The routine has procedural cohesion because it puts a set of operations in a specified order, but the operations don't need to be combined for any other reason.

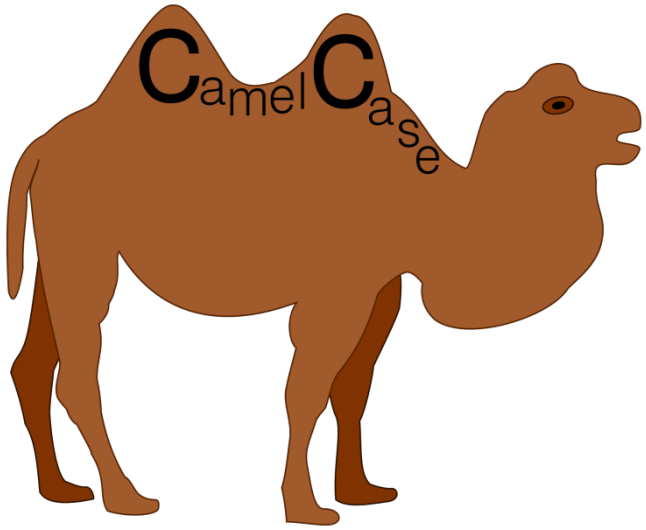
Good Routine Names

Guidelines for creating effective routine names:

- Clearly describe everything the routine does.
  - In the routine's name, describe all the outputs and side effects.
  - If a routine computes report totals and opens an output file, ComputeReportTotals() is not an adequate name for the routine.
    - ComputeReportTotalsAndOpenOutputFile() is an adequate name but is too long and silly.
    - But the cure is not to use less-descriptive routine names.
  - What would be the cure?

CamelCase

- **CamelCase** (**camel case**) is the practice of writing compound words or phrases such that each word or abbreviation begins with a capital letter.
  - Camel case may start with a capital or, especially in programming languages, with a lowercase letter.
  - Common examples are PowerPoint or iPhone.



http://en.wikipedia.org/wiki/CamelCase

Good Routine Names

- Avoid meaningless or vague verbs.
  - Some verbs are elastic, stretched to cover just about any meaning.
    - Routine names like **HandleCalculation**(), **PerformServices**(), **OutputUser**(), **ProcessInput**(), and **DealWithOutput**() don't tell you what the routines do.
      - At best, these names tell you that the routines have something to do with calculations, services, users, input, and output.
    - The exception would be when the verb "**handle**" was used in the specific technical sense of handling an event.
    - If **HandleOutput**() is replaced with **FormatAndPrintOutput**(), you have a pretty good idea of what the routine does.

Good Routine Names

- Don’t differentiate routine names solely by number.
  - Programmers sometimes use numbers to differentiate routines with names like OutputUser, OutputUser1, and OutputUser2.
  - The numerals at the ends of these names provide no indication of the different abstractions the routines represent, and the routines are thus poorly named.

Good Routine Names

- Make names of routines as long as necessary.
  - Research shows that the optimum average length for a variable name is **nine to 15 characters**.
  - Routines tend to be more complicated than variables, and good names for them tend to be longer.
  - On the other hand, routine names are often attached to object names, which essentially provides part of the name for free.
  - Overall, the emphasis when creating a routine name should be to make the name as clear as possible, which means you should make its name as long or short as needed to make it understandable.

Good Routine Names

- To name a function, use a description of the return value.
  - A function returns a value, and the function should be named for the value it returns.
  - For example, **cos()**, **customerId.Next**(), **printer.IsReady**(), and **pen.CurrentColor**() are all good function names that indicate precisely what the functions return.

Good Routine Names

- To name a procedure, use a strong verb followed by an object.
  - The name should reflect what the procedure does, and an operation on an object implies a verb-plus-object name:
    - PrintDocument(), CalcMonthlyRevenues(), CheckOrderInfo(), RepaginateDocument() ,…
  - However, in object-oriented languages, you don't need to include the name of the object in the procedure name because the object itself is included in the call.
    - **document.PrintDocument()** is redundant.
    - If **Check** is a class derived from **Document**, **check.Print()** seems clearly to be printing a check, whereas check.PrintDocument() does not.

Good Routine Names

- Use opposites precisely.
  - Using naming conventions for opposites helps consistency, which helps readability.
    - Opposite-pairs like **first** and **last** are commonly understood.
    - Opposite-pairs like **FileOpen**() and **_lclose**() are not symmetrical and are confusing.
  - Some common opposites:

add/remove	increment/decrement	open/close
begin/end	insert/delete	show/hide
create/destroy	lock/unlock	source/target
first/last	min/max	start/stop
get/put	next/previous	up/down
get/set	old/new	

How Long Can a Routine Be?

- The theoretical best maximum length is often described as one or two pages of program listing, **66 to 132 lines**.
  - In this spirit, IBM once limited routines to 50 lines (McCabe 1976).
  - A large percentage of routines in object-oriented programs will be **accessor routines** (getters and setters), which will be very short.
  - From time to time, a complex algorithm will lead to a longer routine, and in those circumstances, the routine should be allowed to grow organically up to **100 to 200** lines.
  - Decades of evidence say that routines of such length are no more error prone than shorter routines.
  - However, if you want to write routines longer than about 200 lines, be careful.

SYRACUSE UNIVERSITY
ENGINEERING & COMPUTER SCIENCE

How to Use Routine Parameters

Week 9: More Software Implementation Issues

Edmund Yu, PhD
Associate Professor
esyu@syr.edu

How to Use Routine Parameters

- Interfaces between routines are some of the most error-prone areas of a program.
  - One often-cited study by Basili and Perricone (1984) found that 39 percent of all errors were internal interface errors—errors in communication between routines.

Guidelines

- Put parameters in input-modify-output order.
  - Instead of ordering parameters randomly or alphabetically, list the parameters that are input-only first, input-and-output second, and output-only third.
  - This ordering implies the sequence of operations happening within the routine—inputting data, changing it, and sending back a result.

How to Use Routine Parameters

- If several routines use similar parameters, put the similar parameters in a consistent order.
  - The order of routine parameters can be a mnemonic, and inconsistent order can make parameters hard to remember.
    - In C, the **fprintf()** routine is the same as the **printf()** routine except that it adds a file as the first argument. A similar routine, **fputs()**, is the same as **puts**() except that it adds a file as the last argument—BAD.
    - On the other hand, **strncpy**() and **memcpy()** take the same arguments in the same order: target string, source string, and maximum number of bytes—GOOD.

How to Use Routine Parameters

- Use all the parameters.
  - If you pass a parameter to a routine, use it. If you aren't using it, remove the parameter from the routine interface.
  - Unused parameters are correlated with an increased error rate.
    - In one study, **46 percent** of routines with no unused variables had no errors, and only **17 percent to 29 percent** of routines with more than one unused variable had no errors (Card, Church, and Agresti 1986).

How to Use Routine Parameters

- Document assumptions about parameters
  - If you assume the data being passed to your routine have certain characteristics, document the assumptions as you make them.
  - Don’t wait until you’ve written the routine to go back and write the comments.
    - You won’t remember all your assumptions.

How to Use Routine Parameters

- What kinds of interface assumptions about parameters should you document?
  - Whether parameters are input-only, modified, or output-only
  - Units of numeric parameters (inches, feet, meters, and so on)
  - Meanings of status codes and error values
  - Ranges of expected values
  - Specific values that should never appear

How to Use Routine Parameters

- Document interface assumptions about parameters.
  - Even better than commenting your assumptions, use **assertions** to put them into code.

Assertions

- An assertion is code that's used during development that allows a program to check itself as it runs.
  - When an assertion is true, that means everything is operating as expected.
  - When it's false, that means it has detected an unexpected error in the code.
  - For example:
    - If the system assumes that a customer information file will never have more than 50,000 records, the program might contain an assertion that the number of records is less than or equal to 50,000. As long as the number of records is less than or equal to 50,000, the assertion will be silent. If it encounters more than 50,000 records, however, it will loudly "assert" that an error is in the program.

Assertions

- An assertion usually takes two arguments:
  - A **Boolean expression** that describes the assumption that's supposed to be true
  - A **message** to display if it isn't.

- A Java assertion:

```
assert denominator != 0 : "denominator is unexpectedly equal to 0.";
```

- A C++ assertion:

```
assert(denominator != 0);
```

- A Python assertion:

```
assert denominator != 0,  "denominator is unexpectedly equal to 0."
```

How to Use Routine Parameters

- Limit the number of a routine's parameters to about seven.
  - Psychological research has found that people generally cannot keep track of more than about seven chunks of information at once (Miller 1956).
    - This discovery has been applied to an enormous number of disciplines, and it seems safe to conjecture that most people can't keep track of more than about seven routine parameters at once.
  - In practice, how much you can limit the number of parameters depends on how your language handles complex data types.
    - If you program in a modern language that supports structured data, you can pass a composite data type containing 13 fields.

How to Use Routine Parameters

- If you find yourself consistently passing more than a few arguments, the coupling among your routines is too tight.
  - (Re)design the routine or group of routines to reduce the coupling.
  - If you are passing the same data to many different routines, group the routines into a class.

SYRACUSE UNIVERSITY
ENGINEERING & COMPUTER SCIENCE