CPSC 5011: Object-Oriented Concepts

Lecture 4: Has-a, Holds-a

Distinction between Relationships

Implications for Class Design

Software Maintainability, Flexibility & Overhead

Types of Relationships

- None
- Uses-a ***Test Palindrome Using Stack***
  - Call object to retrieve data
- Has-a ***A Plane has-an Engine***
  - SubObject integral part of type definition
- Holds-a ***A Student holds-a Calculator***
  - Object contains but is not defined by subObject
- Is-a ***An Intern is-an Employee***
  - Public inheritance: type extension
- Is-implemented-as-a ***A stack is-implemented-as-a Queue***
  - Private inheritance: code reuse
- As-a ***A student-employee is-a Student and an Employee***
  - Multiple inheritance

Relationships

- Containment aka Holds-A
  - subObjects held, as in a container
  - Little or no type dependency
- Composition aka Has-A
  - subObjects part of class/type composition
  - Essential component => type dependency
- Inheritance aka Is-a
  - Class hierachy: Base (parent) and Derived (child) classes
  - subtype alters or augments inherited behavior
  - Built-in (sub)type checking

Uses-A

```
// Sequence Class method to verify Palindrome Using Stack & Queue
bool Sequence::pal()
{
        Stack   s;                  // no lasting association
        Queue   q;
        bool    match = true;

        while   ( !this->queryDone() )
        {
                Type    store;
                store = this->Query();
                s.push(store);
                q.enQ(store);
        }
        while   ( match && !s.empty( ))
                match = s.pop() == q.deQ();
        return  match;
}
// implementation detail, does not affect Sequence type definition
```

Is-A

An Intern is-an Employee

- Public inheritance: type extension
  - Intern has same functionality as employee
- Intern extends inherited interface
  - Intern may function as an employee
- Collection of employees may include interns
  - Heterogeneous collections

```
class CPPIntern:    public Employee      { … };
class CSharpIntern: Employee             { … };
class JavaIntern extends Employee        {…}
```

Is-implemented-as-a

A Stack is-implemented-as-a Queue

- Private inheritance: code reuse
  - Functionality of Queue used internally by Stack class
- Stack class does not support Queue interface
  - No enQ(), deQ()
  - Stack object cannot be used as a Queue object
- Not DIRECTLY supported in Java/C#
  - No private inheritance in Java/C#
  - Cannot close down a class
  - Cannot override & NOP inherited methods
- Use with caution: not extensible
  - Descendants of Stack have no relationship to Queue

```
class Stack: private Queue
{       …       };
```

As-a: Multiple Inheritance

A student-employee is-a Student and an Employee

- Inheritance from two or more parent classes
- Type extension and/or code reuse
- Nagging implementation details
  - Problems when parent classes are not orthogonal
  - Ambiguity and Redundancy
- Not supported by Java/C#
  - Simulate, using interfaces
  - Cannot seamlessly achieve same degree of code reuse

```
class StudentEmploy: public Employee, public Student
{       …       };
```

Structural Design Details

- Cardinality: How many subObjects?
  - 1:1 for inheritance; 1-many for composition by design
  - Variable for containment
- Ownership
  - Child owns parent component: may NOT be released
  - Composing object may stub out/replace subObject
  - None, usually, for containment
- Lifetime
  - 1:1 for inheritance; variable by design for composition
- Association
  - Permanent for inheritance
  - Possibly transient for composition
  - Temporary for containment

Table 6.1 Relationship details: class to subordinate

Relationship	*Association*	*Cardinality*	*Ownership*	*Dependency*	*Replacement*
Composition	Stable	Variable	Transferable	Yes	Yes
Containment	Temporary	Variable	No	No	Not relevant
Inheritance	Permanent	Fixed: 1-1	Implied	Yes	No

Relationships: Design Details

- Different designs yield different
  - control and maintainability
- Cardinality, ownership, lifetime and association
  - Indicate flexibility, stability, and/or extensibility
- Overhead gauged by these measures.
- For example:
  - has-a relationship may provide varying cardinality
  - is-a relationship cannot vary cardinality
  - has-a may postpone subObject instantiation
  - is-a cannot postpone parent instantiation

Holds-a (Containment)

- Standard Containers
  - Stack, queue, dictionary, hash table
  - May contain subObjects, copies or references
  - number of subObjects may vary within lifetime
- Object Type not defined by subObject
  - *Stack well-defined when empty, full or in-between*
  - subObject provides no direct (public interface) functionality for container
- subObject(s) not necessarily owned by container
  - May be shareable
  - May be passed in, passed out

A Student holds-a Calculator

```
class AvgStudent                    // replaceable calculator
{       Calculator*     c;
        …
};

class PrecisionStudent              // zero, one or more calculators
{       Calculator*     c;
        int             numCalc;
        …
};

class SharingStudent    // calculator shared with others
{                       // calculator has associated reference count
        Calculator*     c;
        …
};

// C#?  CONSIDER REPLACEABILITY
```

```
class AvgStudent {
       Calculator*   c;
   public:
       …
       // assumption constructor:
       //     ownership of transfer assumed
       AvgStudent(Calculator *& transfer) {
              c = transfer;
              transfer = 0;
       }

       // ownership implies memory management
       //     => destructor, copy constructor,
       //              overloaded assignment operator
};

// FORM OF A replace() FUNCTION???
// NOTION OF OWNERSHIP IN C#
```

```
class PrecisionStudent {
       Calculator*   c;
       int           numCalc;
   public:
       …
       PrecisionStudent()
       {      c = numCalc = 0;     }

       // if needed, subObjects generated internally
       PrecisionStudent(unsigned   quantity) {
              numCalc = quantity;
              c = new Calculator[numCalc];
       }

       // ownership implies memory management
       //     => destructor, copy constructor,
       //              overloaded assignment operator
};
```

Disposal of SubObjects

SubObjects temporarily owned by Container

- Invoke subObject destructor
  - Will deallocate if heap and sole owner
  - Will reduce reference count if ownership shared
- Pass Out Ownership
  - Functions like dequeue() return subObject
  - Internal storage (pointer) zeroed

SubObjects not owned

- Zero out internal storage

A Student holds-a Calculator

- Containment
- Core student functionality not defined by calculator
- Student well-defined without calculator
  - State of calculator does not *drive* state of student
  - Student may have zero calculators and still function as a student
- Student may or may not own calculator
  - Student does not hold calculator for lifetime
  - Calculators may be shared, borrowed, transferred, …
  - Student not always responsible for creation/destruction of calculator(s)
- A given student may hold varying numbers of calculators
  - Cardinality varies over lifetime of student

Key Design Decision

- Suppress Copying
  - Declare copy constructor private
- Support (Deep) Copying
  - Declare copy constructor public
  - Define copy constructor
- NO DECISION => Aliasing (Shallow Copy)
  - Compiler-provided copy constructor used
  - DANGER
- C++ 11: MOVE SEMANTICS
  - Use '&&' to flag compiler
  - Copying suppressed by compiler
    - When appropriate (TEMPORARIES)
    - Enhances performance (does not yield persistent unintended aliases)

```
class AvgStudent    // no copying supported
{
       Calculator*   c;

       // declare copy constructor private
       //     => cannot be invoked outside class methods
       AvgStudent(const AvgStudent*&);
   public:
       …
};

// application code segments with compile-time errors
void passByValue(AvgStudent a)    {       …       }
AvgStudent  returnByValue()               {       …       }

AvgStudent    a;
AvgStudent    b(a);
```

What about Assignment?

Valid to copy one container's content to another?

- Support Copying Content
  - Declare and define overloaded assignment operator
- Suppress Copying Content
  - Like ADA's Limited type construct
  - Declare overloaded assignment operator private

```
class AvgStudent    // assignment not supported
{
       Calculator*   c;

   // declare overloaded assignment operator private
       //     => cannot be invoked outside class methods
       void operator=(const AvgStudent*&);
   public:
       …
};

// application code segments with compile-time errors
   Calculator        *x = new Calculator(10);
   Calculator *y = new Calculator;

   AvgStudent        a(x);
   AvgStudent b(y);

   b = a;
```

Has-a (Composition)

- Object Type defined by subObject(s)
  - Data membership
    - Typically instantiated upon object construction
    - Cardinality usually fixed within lifetime
  - Correlation between lifetimes
  - subObject provides functionality
  - subObject affects state of Object

- subObject(s) owned
  - Not usually shareable or transferable
  - May be replaceable
  - Object may be responsible for allocation/deallocation

A Plane has-an Engine

```
// plane type must have engine(s):  Flexible?
class PlaneA1
{
      Engine        e;
  public:
      …
};

class PlaneA2
{
      Engine        e[4];
  public:
      …
};
```

```
class PlaneA1 // initializer list
{
        // when PlaneA1 object constructed
        //     default constructor for Engine invoked
        //     UNLESS class designer overrides
        Engine e;
   public:
        // default constructor for subObject invoked
        PlaneA1();

   // initializer list specifies subObject constructor
        //     => Engine(int) constructor invoked
        PlaneA1(int x): e(x)
        …
};
```

```
class PlaneA2 // subObject replacement
{
        // when PlaneA1 object constructed
        //      default constructor for Engine invoked
        //      cannot override array allocation
        Engine e[4];
   public:
        // default construction of engines ok
        PlaneA2();

        // Engine subObjects constructed before PlaneA2
        //      => replace (copy) engine with new engine
        PlaneA2(int* p)
        {       Engine temp(p[0]);
                e[0] = temp;
                …
        }
        …
};
```

A Plane has-an Engine

- Composition
- Plane not well-defined without engine
  - Plane not usable without engine
  - State of engine affects state of plane

- Plane owns its engine
  - Lifetime association between engine & plane
  - Engine not shareable with other planes
  - Plane responsible for creation/destruction of engine(s)
  - Engine may be replaced by another engine

- Engine provides needed functionality
- A given plane has a specific number of engines
  - Cardinality fixed, typically for lifetime of plane

Details

- Determine copying & assignment legality
  - Deep or shallow copying? Move semantics?
- Relationship determined by design
  - Syntax does not drive relationship
  - subObjects could be held by pointers
- Consider replacement carefully
  - Is null an option?
    - Only when object first constructed?
  - Control replacement
    - place conditions & test

Table 6.1 Relationship details: class to subordinate

Relationship	*Association*	*Cardinality*	*Ownership*	*Dependency*	*Replacement*
Composition	Stable	Variable	Transferable	Yes	Yes
Containment	Temporary	Variable	No	No	Not relevant
Inheritance	Permanent	Fixed: 1-1	Implied	Yes	No

Elements of Relationship

- Lifetime of subObject relative to object
  - 1-1 correspondence implies has-a
- Ownership of subObject
  - Owned
  - Shared
  - Referenced
- Association
  - Permanent implies has-a
  - Temporary
    - Replaceable
    - Possibly null

Relationships: Design Details

- Different designs yield different
  - control and maintainability
- Cardinality, ownership, lifetime and association
  - Indicate flexibility, stability, and/or extensibility
- Overhead gauged by these measures.
- For example:
  - has-a relationship may provide varying cardinality
  - is-a relationship cannot vary cardinality
  - has-a may postpone subObject instantiation
  - is-a cannot postpone parent instantiation

Example 6.1 Postponed Instantiation of SubObject

```
class justInTime
{       // need appropriate memory management details
        //      Suppress or define: copy constructor and operator=
        bigData*        generator;
        …
   public:

        justInTime()    {       generator = 0; }
        …
        void process()
        {
                if ( !generator )       generator = new bigData;
                generator.process();
        }
        …
};
```

Association

- Uses-a

  Test Palindrome Using Stack

  – Temporary: function call (*message invocation*)

  – Longer: used object embedded => holds-a or has-a relation

- Is-a

  An Intern is-an Employee

  – Permanent: required, non-optional overhead

  – Lifetime Association, whether used or not

- Is-implemented-as-a

  A stack is-implemented-as-a Queue

  – Same as above

  – Key difference: interface of parent automatically suppressed

Association

- Has-a

  A Plane has-an Engine

  – Lifetime:
    - subObject provides essential functionality for object

  – May be constructed at will
    - subObject instantiated with Object
    - subObject instantiated upon first use

- Holds-a

  A Student holds-a Calculator

  – Ephemeral:
    - subObject(s) may go in & out of containment

What type of relationship?

- A student has a Ferrari
- A book has a bookmark
- A book has a ISBN number
- A CPSC class has a student

Has-A vs. Holds-a

- Has-a relationships imply type dependency
  - class dependent on subObject for data and/or functionality
  - Constructor may instantiate subObject
  - Class methods possible for replacement
- Holds-a relationships imply temporary association
  - No significant dependency on type held
    - Type could be replaced
    - Number of subObjects held could be zero, without impact
  - type independence like that of a container.

Class Design

When an object contains one or more subobjects, consider:

- Ownership
  - Are the subobject(s) shared with other objects?
  - Can ownership be transferred?
    - Passed off to another object rather than deleted
    - Assumed from another object rather than constructed
- Cardinality
  - Is number of subobjects determined upon construction?
  - Is number of subobjects fixed?
- Association

Ownership Implies

- Access
  - Object has access to public interface of subObject
    - Application Programmer has no access to subObject
    - Has-a does not give access to protected interface of subObject
  - Object may echo subObject public interface
- Control
  - May replace or discard subObject
  - *Avoid overhead of subObject*
    - *null reference or pointer*
- Responsibility
  - Cleanup (constructor, destructor, …)
  - Consistency
    - state of subObject
    - Beware redundancy

C++ vs. C# Class Design

- C++ classes with internal heap memory must provide destructors.
- C# relies on implicit deallocation (garbage collection)
  - classes do not need destructors.
- Both languages should make explicit decisions with respect to copying.
- C# class designers, for deep copying,
  - implement a Cloneable interface
  - require client to cast the object to the appropriate type
  - by default, copying is shallow.
- C++ class designers may
  - suppress copying, support deep copying or employ move semantics.

Principle of Least Knowledge

- *Every object should assume the minimum possible about the structure and properties of other objects.*
- Promotes low coupling
- when classes interact, in any relationship
  - class design should not be dependent on private implementation details of any other class.
- With clear documentation, deliberate design identifies relationships and their consequential effects.

Structured vs. OO

- Conceptual difference
  - emphasis
- Technical difference
  - Language constructs built-in for inheritance
- Run-time difference
  - Dynamic function invocation

Professional Goals

- Understand design costs & benefits
  - choose among alternatives
  - identify performance impacts
- Identify basic concepts
  - simulate missing features
  - avoid expensive approaches
- Use design appropriately