SOFTENG 701: Advanced Software Engineering Development Methods Part 2

Lecture 2b: Johnson and Foote on Reusable Designs

Ewan Tempero
Department of Computer Science

Agenda

- Understanding reusability
- Ralph E. Johnson and Brian Foote *Designing Reusable Classes* Journal of Object-Oriented Programming June/July 1988, Volume 1, Number 2, pages 22-35

- Agenda
- Introduction
- Reusable vs. Reuse
- Context Reuse
- Reusable Designs
- Example
- Abstract Classes
- Frameworks
- Key Points

Design Advice for how to create reusable designs

- Quality Attribute = “Reusability”
- “object-oriented” = (in order) Objects, Polymorphism, “Protocol”, Inheritance, Abstract Classes (i.e. Smalltalk)
- Johnson and Foote, early (1988) discussion on how “object-oriented design” can support “reusability”
- Also early discussion on frameworks and refactoring
- Says “Object-oriented programming is often touted as promoting software reuse”, but observes that reusability does not just happen, “Program components must be designed for reusability”
- How do Johnson and Foote define “reusability”? — They don’t!
    - ○ tendency towards “number of places component can be used” but also elements of “ease of reuse”

Reusable vs. Reuse

- Reuse — create a new thing without having to create everything it's made of (by reusing existing things)
  - Inheritance-as-implementation supports reuse
  - So does aggregation (aka composition), perhaps even more so
  - So does cut'n'paste . . .
- Reusable — ease with which something can be reused to create a new thing
  - Inheritance-as-subtype supports reusability
- Questions:
  - How is "ease" measured?
  - Does "ease" include "how easy is it to understand how to reuse it"?
  - Does "ease" include "effort required to assure that the result is correct (aka testing)
  - Does "ease" mean "time saved compared with not reusing and writing from scratch"?

- Agenda
- Introduction
- Reusable vs. Reuse
- Context Reuse
- Reusable Designs
- Example
- Abstract Classes
- Frameworks
- Key Points

Context Reuse

```
class Person {
  public void display () { ... }
  ...
}
class Student extends Person {
  public void display () { ... }
  ...
}
class Tutor extends Person {
  public void display () { ... }
  ...
}
class Client {
  public void display(List<Person> list) {
    for (Person p: list) {
      p.display();
    }
  }
}
```

- The thing that is reused should be the thing that is unchanged
- The code that doesn't change here is the context
- Code is reusable because only objects that have the `display()` method are presented to it

- Agenda
- Introduction
- Reusable vs. Reuse
- Context Reuse
- Reusable Designs
- Example
- Abstract Classes
- Frameworks
- Key Points

Johnson and Foote on Reusable Designs

- Standard Protocols

  Rule 1: Recursion introduction
  Rule 2: Eliminate case analysis
  Rule 3: Reduce the number of arguments
  Rule 4: Reduce the size of methods

- Abstract Classes

  Rule 5: Class hierarchies should be deep and narrow
  Rule 6: The top of the class hierarchy should be abstract
  Rule 7: Minimise access to variables
  Rule 8: Subclasses should be specialisations

- Frameworks

  Rule 9: Split large classes
  Rule 10: Factor implementation differences into subcomponents
  Rule 11: Separate methods that do not communicate
  Rule 12: Send messages to components instead of to `self`
  Rule 13: Reduce implicit parameter passing

- Agenda
- Introduction
- Reusable vs. Reuse
- Context Reuse
- Reusable Designs
- Example
- Abstract Classes
- Frameworks
- Key Points

Standard Protocols

- Protocol — the set of messages that can be send to an object
- Objects with the same protocol can be substituted for each other

⇒ more things with the same protocol means more reusable

- Subtypes share their parent type's protocol

⇒ inheritance as subtype

Rules for Finding Standard Protocols

- Rule 1: Recursion introduction
  - Using the same names for operations means protocols are more likely to be the same
- Rule 2: Eliminate case analysis
  - Checks for what kind of object is being processed can be replaced by messages to subtypes — which all have the same protocol
- Rule 3: Reduce the number of arguments
  - Few arguments means operations will more likely look like others
  - Reduce arguments by packaging arguments in a class†
- Rule 4: Reduce the size of methods
  - Classes with small methods are "easier to subclass"
  - "A thirty line method is large"
  - Eliminating cases usually leads to smaller methods
  - Making a method smaller will likely reduce the number of arguments

Rule 2 (Eliminate case analysis) Example

- Agenda
- Introduction
- Reusable vs. Reuse
- Context Reuse
- Reusable Designs
- Example
- Abstract Classes
- Frameworks
- Key Points

```
static final int MONSTER = 1;
static final int CIVILIAN = 2;
static final int SOLDIER = 3;
...

void process(int u) {
  if (u == MONSTER) {
    // do monster stuff
  } else if (u == CIVILIAN) {
    // do civilian stuff
  } ...
}
```

```
// Make sure all classes have same protocol (‘‘Unit’’)
class Monster implements Unit { ... }
class Civilian implements Unit { ... }

...
void process (Unit u) {
  u.doStuff();
}
```

Questions

- How often are methods names the same? (recursion introduction)
- Is the cost of removing case analysis justified by the savings? What are the savings really?
- How many parameters do methods typically have?
- Are there many big methods? Is the cost of reducing their size worth it?

Abstract Classes

- Classes that do not have complete implementations and so cannot be instantiated
- Subclasses of abstract classes share the protocol of their parents
- Roots of class hierarchies tend to be abstract, leaves cannot
- Can provide program “skeletons”
- “it is better to inherit from an abstract class than from a concrete class”

Rules for Finding Abstract Classes

- Rule 5: Class hierarchies should be deep and narrow
  - “one superclass and 27 subclasses is much too shallow”
- Rule 6: The top of the class hierarchy should be abstract
- Rule 7: Minimise access to variables
  - “Classes can be made more abstract by eliminating their dependence on their data representation”
  - ⇒ use getters and setters
- Rule 8: Subclasses should be specialisations
  - Inheritance-as-subtype

Questions

- What proportion of types are abstract?
- What proportion of types have abstract types as ancestors?
- When inheritance is used to define a type, what proportion of the defined types inherit from abstract types?
- What is the cost of inheritance from a non-abstract type?
- What is the cost of shallow hierarchies?

- Agenda
- Introduction
- Reusable vs. Reuse
- Context Reuse
- Reusable Designs
- Example
- Abstract Classes
- Frameworks
- Key Points

Frameworks

- A collection of types providing some complex functionality
- A bigger unit of reuse than classes
- A “skeleton” for functionality
- Includes (but not limited to) abstract classes
- Types refer to each other but otherwise self-contained
- Examples: AWT, Swing, Spring, (Java Collections Framework)

Rules for Finding Frameworks

- Rule 9: Split large classes
  - A class represents an abstraction ⇒ a large class is probably several abstractions the work together
- Rule 10: Factor implementation differences into subcomponents
  - Some subclasses with common method implementations may be refactored with a common superclass
- Rule 11: Separate methods that do not communicate
  - Groups of methods that don’t share fields can be in separate classes (cohesion)
- Rule 12: Send messages to components instead of to `self`
  - Rather than relying on inheritance-base self-calls to provide variation, create components representing the variations and use polymorphic calls to them
- Rule 13: Reduce implicit parameter passing
  - Avoid passing information from one method to another via a field, instead pass as an explicit parameter

Questions

- How many big classes exist? How must does it cost to split them?
- How often are are there methods that do not communicate? What is the cost of splitting such classes?
- How often does implicit parameter passing occur? What is the cost of removing it?

- Agenda
- Introduction
- Reusable vs. Reuse
- Context Reuse
- Reusable Designs
- Example
- Abstract Classes
- Frameworks
- Key Points

Key Points

- Concepts in the object-oriented paradigm are considered to provide better support for reusability (and understandability) than what came before
- Many ideas for “reusable” design have been around for a long time