# GRASP: More Objects with Responsibilities

Dalibor Dvorski (ddvorski@conestogac.on.ca)

School of Engineering and Information Technology

Conestoga College Institute of Technology and Advanced Learning

# Polymorphism

*Problem*: How to handle alternatives based on type?  How to create pluggable software components?

- *"alternatives based on type"*: Conditional variation is a fundamental theme in programs.  If a program is designed using if-then-else or case statement conditional logic, then if a new variation arises, it requires modification of the case logic – often in many places.  This approach makes it difficult to easily extend a program with new variations because changes tend to be required in several places – whenever the conditional logic exists.
- *"pluggable software components"*: Viewing components in client-server relationships, how can you replace one server component with another, without affecting the client?

*Solution*: When related alternatives or behaviours vary by type (class), assign responsibility for the behaviour – using polymorphic operations – to the types for which the behaviour varies.

- *Corollary*: Do not test for the type of an object and use conditional logic to perform varying alternatives based on type.

## Polymorphism

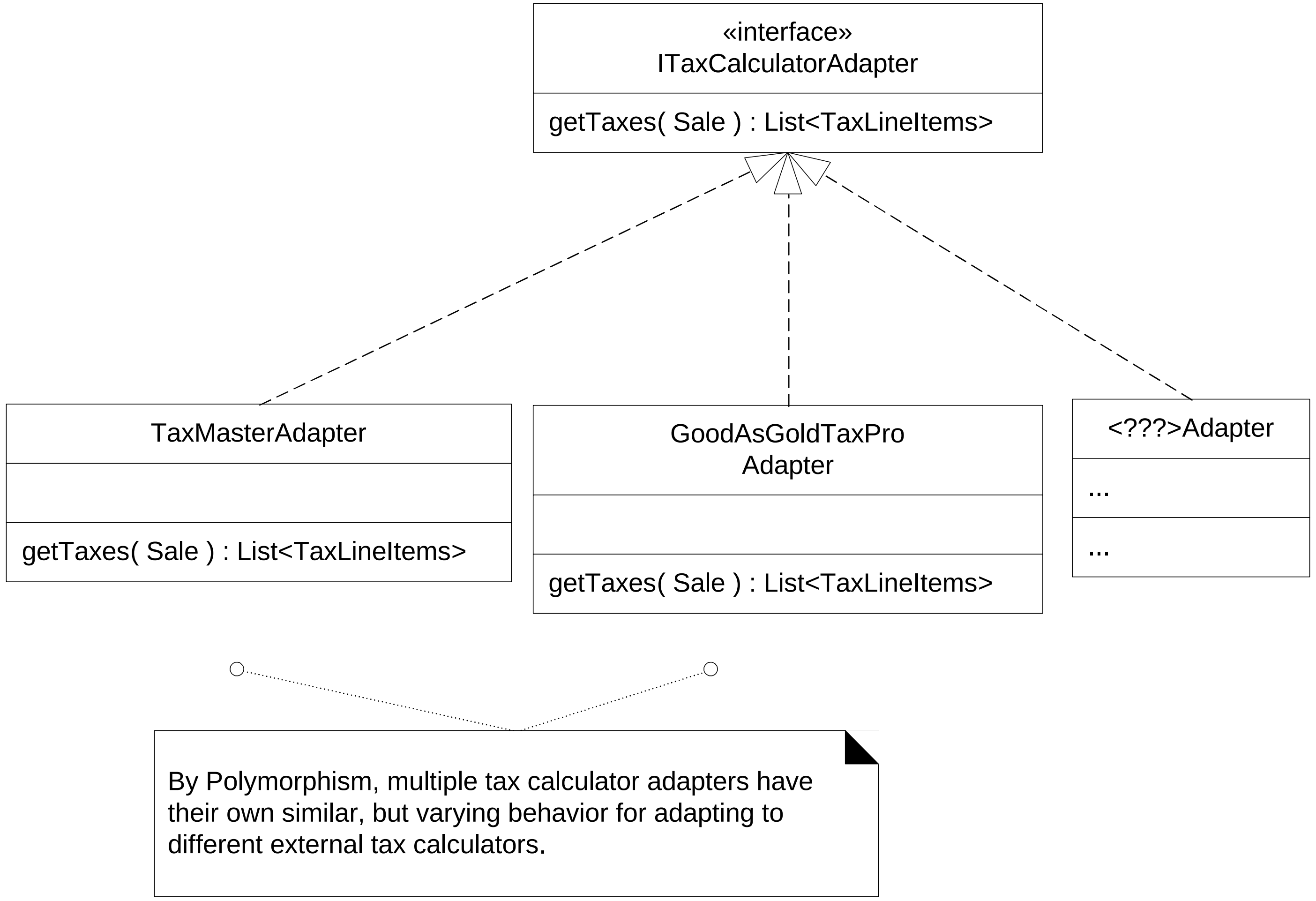
e.g. In the NextGen POS application, there are multiple external third-party tax calculators that must be supported (e.g. Tax-Master, GoodAsGoldTaxPro); the system needs to be able to integrate with different ones.  Each tax calculator has a different interface, so there is similar but varying behaviour to adapt to each of these external fixed interfaces or application programming interfaces (APIs).

What objects should be responsible for handling these varying external tax calculator interfaces?

Since the behaviour of calculator adaptation varies by the type of calculator, by Polymorphism we should assign the responsibility for adaptation to different calculator objects themselves, implemented with a polymorphic `getTaxes` operation.

These calculator adapter objects are not the external calculators, but rather, local software objects that represent the external calculators, or the adapter for the calculator.  By sending a message to the local object, a call will ultimately be made on the external calculator in its native API.

# Polymorphism


«interface»
ITaxCalculatorAdapter
getTaxes( Sale ) : List<TaxLineItems>
TaxMasterAdapter
getTaxes( Sale ) : List<TaxLineItems>
GoodAsGoldTaxPro Adapter
getTaxes( Sale ) : List<TaxLineItems>
<???>Adapter
...
...

By Polymorphism, multiple tax calculator adapters have their own similar, but varying behavior for adapting to different external tax calculators.

Each `getTaxes` method takes the `Sale` object as a parameter, so that the calculator can analyze the sale. The implementation of each `getTaxes` method will be different: `TaxMasterAdapter` will adapt the request to the API of Tax-Master, etc.

Polymorphism is a fundamental principle in designing how a system is organized to handle similar variations. A design based on assigning responsibilities by Polymorphism can be easily extended to handle new variations. e.g. Adding a new calculator adapter class with its own polymorphic `getTaxes` method will have minor impact on the existing design.

## Polymorphism

*When to design with interfaces?*

Polymorphism implies the presence of abstract superclasses or interfaces in most OO languages. When should you consider using an interface? The general answer is to introduce one when you want to support polymorphism without being committed to a particular class hierarchy. If an abstract superclass `AC` is used without an interface, any new polymorphic solution must be a sublass of `AC`, which is very limiting in single-inheritance languages such as C# and Java. As a rule-of-thumb, if there is a class hierarchy with an abstract superclass `C1`, consider making an interface `I1` that corresponds to the public method signatures of `C1`, and then declare `C1` to implement the `I1` interface. Then, even if there is no immediate motivation to avoid subclassing under `C1` for a new polymorphic solution, there is a flexible evolution point for unknown future cases.

# Polymorphism

*Contraindications*: Sometimes, developers design systems with interfaces and polymorphism for speculative "future-proofing" against an unknown possible variation. If the variation point is definitely motivated by an immediate or very probable variability, then the effort of adding the flexibility through polymorphism is of course rational. But critical evaluation is required, because it is not uncommon to see unnecessary effort being applied to future-proofing a design with polymorphism at variation points that in fact are improbable and will never actually arise. Be realistic about the true likelihood of variability before investing in increased flexibility.

## Pure Fabrication

*Problem*: What object should have the responsibility, when you do not want to violate High Cohesion and Low Coupling, or other goals, but solutions offered by Expert (for example) are not appropriate?

Object-oriented designs are sometimes characterized by implementing as software classes representations of concepts in the real-world problem domain to lower the representational gap; e.g. a `Sale` and `Customer` class. However, there are many situations in which assigning responsibilities only to domain layer software classes leads to problems in terms of poor cohesion or coupling, or low reuse potential.

*Solution*: Assign a highly cohesive set of responsibilities to an artificial or convenience class that does not represent a problem domain concept – something made up, to support high cohesion, low coupling, and reuse.

Such a class is a fabrication of the imagination. Ideally, the responsibilities assigned to this fabrication support high cohesion and low coupling, so that the design of the fabrication is very clean, or pure – hence a pure fabrication.

## Pure Fabrication

e.g. Suppose that support is needed to save `Sale` instances in a relational database. By Information Expert, there is some justification to assign this responsibility to the `Sale` class itself, because the sale has the data that needs to be saved. But consider the following implications:

- The task requires a relatively large number of supporting database-oriented operations, none related to the concept of sale-ness, so the `Sale` class becomes incohesive.
- The `Sale` class has to be coupled to the relational database interface, so its coupling goes up. And the coupling is not even to another domain object, but to a particular kind of database interface.
- Saving objects in a relational database is a very general task for which many classes need support. Placing these responsibilities in the `Sale` class suggests there is going to be poor reuse or lots of duplication in other classes that do the same thing.

Thus, even though `Sale` is a logical candidate by virtue of Information Expert to save itself in a database, it leads to a design with low cohesion, high coupling, and low reuse potential – exactly the kind of desperate situation that calls for making something up.

# Pure Fabrication

A reasonable solution is to create a new class that is solely responsible for saving objects in some kind of persistent storage medium, such as a relational database; call it `PersistentStorage`. This class is a Pure Fabrication – a figment of the imagination.

Notice the name: `PersistentStorage`. This is an understandable concept, yet the name or concept “persistent storage” is not something one would find in the Domain Model. And if a designer asked a business-person in a store, “Do you work with persistent storage objects?” they would not understand. They understand concepts such as “sale” and “payment”. `PersistentStorage` is not a domain concept, but something made up or fabricated for the convenience of the software developer.

This Pure Fabrication solves the following design problems:

- The `Sale` remains well-designed, with high cohesion and low coupling.
- The `PersistentStorage` class is itself relatively cohesive, having the sole purpose of storing and inserting objects in a persistent storage medium.
- The `PersistentStorage` class is a very generic and reusable object.

## Pure Fabrication

*Contraindications*: Behavioural decomposition into Pure Fabrication objects is sometimes overused by those new to object design and more familiar with decomposing or organizing software in terms of functions. To exaggerate, functions just become objects. There is nothing inherently wrong with creating “function” or “algorithm” objects, but it needs to be balanced with the ability to design with representational decomposition, such as the ability to apply Information Expert so that a representational class such as `Sale` also has responsibilities. Information Expert supports the goal of co-locating responsibilities with the objects that know the information needed for those responsibilities, which tends to support lower coupling. If overused, Pure Fabrication could lead to too many behaviour objects that have responsibilities not co-located with the information required for their fulfillment, which can adversely affect coupling. The usual symptom is that most of the data inside the objects is being passed to other objects to reason with it.

# Indirection

*Problem*: Where to assign a responsibility, to avoid direct coupling between two (or more) things? How to de-couple objects so that low coupling is supported and reuse potential remains higher?

*Solution*: Assign the responsibility to an intermediate object to mediate between other components or services so that they are not directly coupled.

e.g. `TaxMasterAdapter` and `GoodAsGoldTaxProAdapter` act as intermediaries to the external tax calculators. Via polymorphism, they provide a consistent interface to the inner objects and hide the variations in the external APIs. By adding a level of indirection and adding polymorphism, the adapter objects protect the inner design against variations in the external interfaces.

e.g. The Pure Fabrication example of decoupling the `Sale` from the relational database services through the introduction of a `PersistentStorage` class is also an example of assigning responsibilities to support Indirection. The `PersistentStorage` acts as an intermediary between the `Sale` and the database.

# Protected Variations

*Problem*: How to design objects, subsystems, and systems so that the variations or instability in these elements does not have an undesirable impact on other elements?

*Solution*: Identify points of predicted variation or instability; assign responsibilities to create a stable interface around them.

e.g. The prior external tax calculator problem and its solution with Polymorphism illustrate Protected Variations. The point of instability or variation is the different interfaces or APIs of external tax calculators. The POS system needs to be able to integrate with many existing tax calculator systems, and also with future third-party calculators not yet in existence. By adding a level of indirection, an interface, and using polymorphism with various `ITaxCalculatorAdapter` implementations, protection within the system from variations in external APIs is achieved. Internal objects collaborate with a stable interface; the various adapter implementations hide the variations to the external systems.

Protected Variations was first published as a named pattern by Cockburn, although this very fundamental design principle has been around for decades under various terms, such as the term *information hiding*.

## Assigned Readings

1. Chapter 23: Iteration 2 – More Patterns from
   Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development.
2. Chapter 24: Quick Analysis Update from
   Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development.
3. Chapter 25: GRASP: More Objects with Responsibilities from
   Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development.
   - Note the coverage of the Monopoly case study and application of UML.
   - Differentiate between representational decomposition and behavioural decomposition.
   - What are the benefits of each of the discussed design patterns?
   - Read about Protected Variations as a root principle motivating most of the mechanisms and patterns in programming and design.