SE 2AA4, CS 2ME3 (Introduction to Software Development)

Winter 2018

36 Design Patterns DRAFT

Dr. Spencer Smith

Faculty of Engineering, McMaster University

December 15, 2017

36 Design Patterns DRAFT

- Administrative details
- Debugging
- Verifying performance and reliability
- Design patterns

Administrative Details

TBD

Verifying Performance

- Worst case analysis versus average behaviour
- For worst case focus on proving that the system response time is bounded by some function of the external requests
- Standard deviation
- Analytical versus experimental approaches
- Consider verifying the performance of a pacemaker
- Visualize performance via
  - Identify a measure of performance (time, storage, FLOPS, accuracy, etc.)
  - Identify an independent variable (problem size, number of processors, condition number, etc.)

Verifying Reliability

- There are approaches to measuring reliability on a probabilistic basis, as in other engineering fields
- Unfortunately there are some difficulties with this approach
- Independence of failures does not hold for software
- Reliability is concerned with measuring the probability of the occurrence of failure
- Meaningful parameters include
  - Average total number of failures observed at time t: $AF(t)$
  - Failure intensity: $FI(T) = AF'(t)$
  - Mean time to failure at time t: $MTTF(t) = 1/FI(t)$
- Time in the model can be execution or clock or calendar time

Verifying Subjective Qualities

- What do you think is meant by empirical software engineering?
- What problems might be studied by empirical software engineering?
- Does the usual engineering analogy hold for empirical software engineering?

Verifying Subjective Qualities

- Consider notions like simplicity, reusability, understandability
- Software science (due to Halstead) has been an attempt
- Tries to measure some software qualities, such as abstraction level, effort,
- by measuring some quantities on code, such as
  - η_1, number of distinct operators in the program
  - η_2, number of distinct operands in the program
  - N_1, number of occurrences of operators in the program
  - N_2, number of occurrences of operands in the program
- Extract information from repo, including number of commits, issues etc.
- Empirical software engineering
- Appropriate analogy switches from engineering to medicine

Source Code Metric

- What are the consequences of complex code?
- How might you measure code complexity?

McCabe's Source Code Metric

- Cyclomatic complexity of the control graph
  - $C = e - n + 2p$
  - e is number of edges, n is number of nodes, and p is number of connected components
- McCabe contends that well-structured modules have C in range 3..7, and $C = 10$ is a reasonable upper limit for the complexity of a single module
- Confirmed by empirical evidence

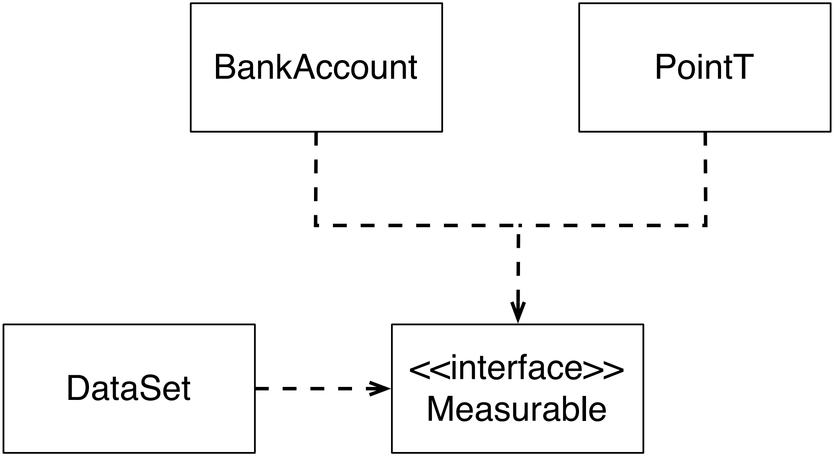
Design Patterns

- Christopher Alexander (1977, buildings/towns):
  - “Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way.”
- Design reuse (intended for OO)
- Solution for recurring problems
- Transferring knowledge from expert to novice
- A design pattern is a recurring structure of communicating components that solves a general design problem within a particular context
- Design patterns consist of multiple modules, but they do not constitute an entire system architecture

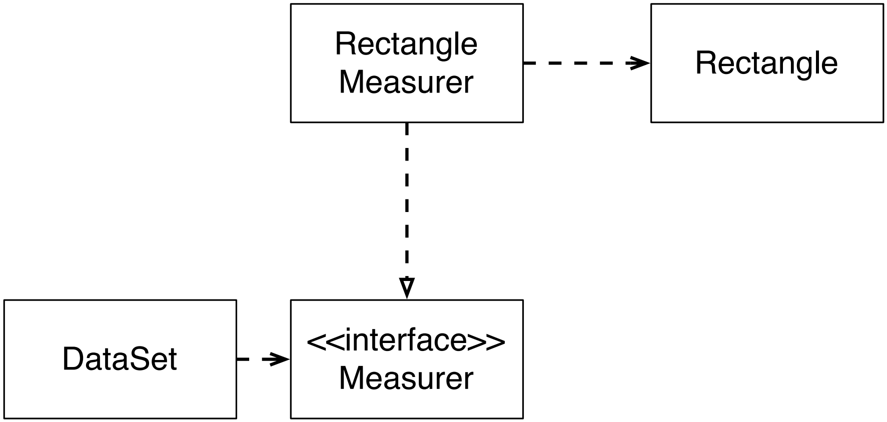
Strategy Design Pattern

- From Source Making web-page
- Define a family of algorithms, encapsulate each one, and make them interchangeable.
- Strategy lets the algorithm vary independently from the clients that use it.
- Capture the abstraction in an interface, bury implementation details in derived classes.
- Where have we used this pattern?

UML Diagram of Measurable Interface



UML Diagram of Measurer Interface



Model View Controller (MVC)

- Separate computational elements from I/O elements
- Three components
  1. Model encapsulates the system's data as well as the operations on the data
  2. View displays the data from the model components, possibly multiple view components
  3. Controller handles input actions
- The controller may or may not depend on the state of the model
- The controller depends on model state when menu items are enabled or disabled depending on the state of the model

Design Pattern Properties

- A pattern addresses a recurring design problem that arises in specific design situations and presents a solution to it
- A pattern must balance a set of opposing forces
- Patterns document existing, well-proven design experience
- Patterns identify and specify abstractions above the level of single components (modules)
- Patterns provide a common vocabulary and understanding for design principles
- Patterns are a means of documentation
- Patterns support the construction of software with defined properties, including non-functional requirements, such as flexibility and maintainability

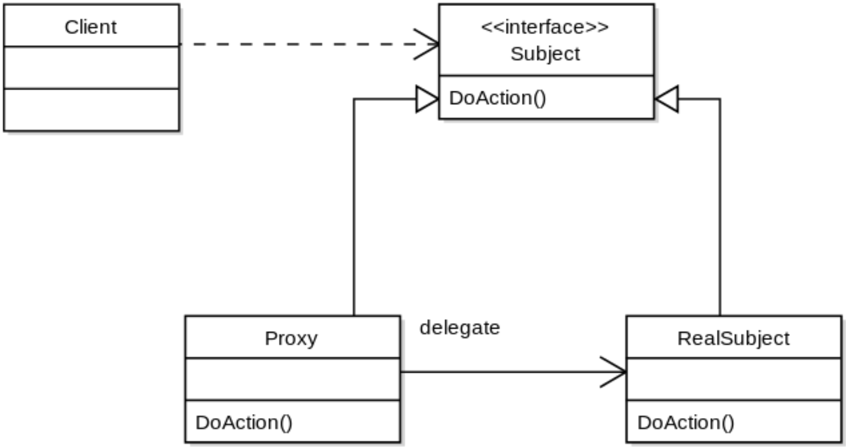
Describing Patterns

- Context: the situation giving rise to a design pattern
- Problem: a recurring problem arising in that situation
- Solution: a proven solution to that problem

The Proxy Pattern (from van Vliet (2000))

- Context: A client needs services from another component. Though direct access is possible, this may not be the best approach
- Problem: We do not want to hard-code access to a component into a client. Sometimes, such direct access is inefficient; in other cases it may be unsafe. This inefficiency or insecurity is to be handled by additional control mechanisms, which should be kept separate from both the client and the component to which it needs access.
- Solution: The client communicates with a representative rather than the component itself. This representative, the proxy, also does and pre- and postprocessing that is needed.
- Code

UML Diagram of Proxy



Command Processor Pattern

- Context: User interfaces which must be flexible or provide functionality that goes beyond the direct handling of user functions. Examples are undo facilities or logging functions
- Problem: We want a well-structured solution for mapping an interface to the internal functionality of a system. All 'extras' which have to do with the way user commands are input, additional commands such as undo and redo, and any non-application-specific processing of user commands, such as logging, should be kept separate from the interface to the internal functionality.

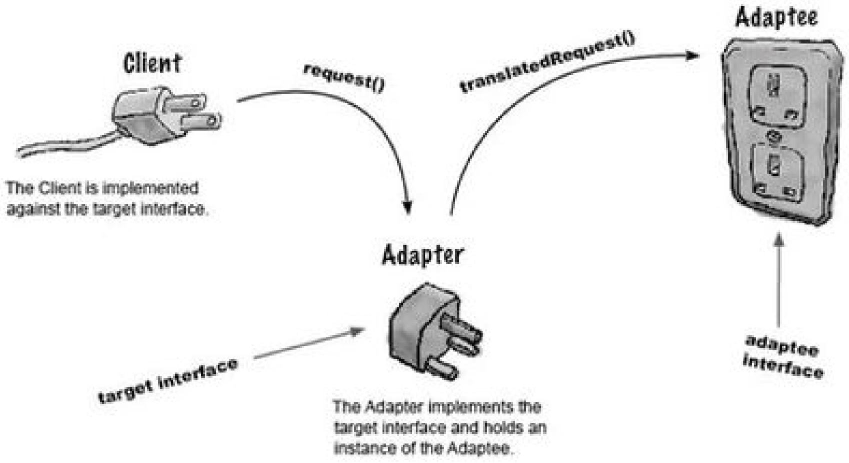
Command Processor Pattern Continued

- Solution: A separate component, the command processor, takes care of all commands. The command processor component schedules the execution of commands, stores them for later undo, logs them for later analysis, and so on. The actual execution of the command is delegated to a supplier component within the application.

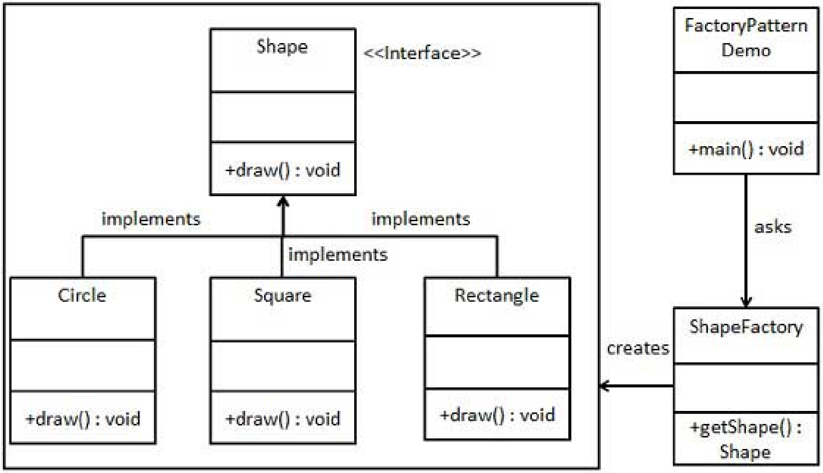
Adapter Design Pattern

When have we used the adapter (or wrapper) design pattern?

Adapter Design Pattern

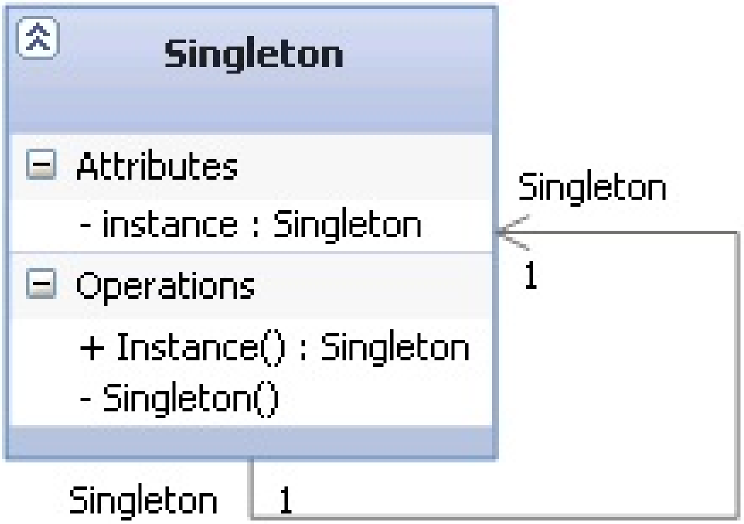

Client
request()
translatedRequest()
Adaptee
The Client is implemented against the target interface.
Adapter
target interface
The Adapter implements the target interface and holds an instance of the Adaptee.
adaptee interface

Factory Pattern



Code

Singleton Pattern


Singleton
Attributes
- instance : Singleton
Operations
+ Instance() : Singleton
- Singleton()
Singleton
1
Singleton
1