Design Patterns: Creational

30 March 2009
CMPT166
Dr. Sean Ho
Trinity Western University

See also:
Vince Huston

TRINITY WESTERN UNIVERSITY

Review last time: testing

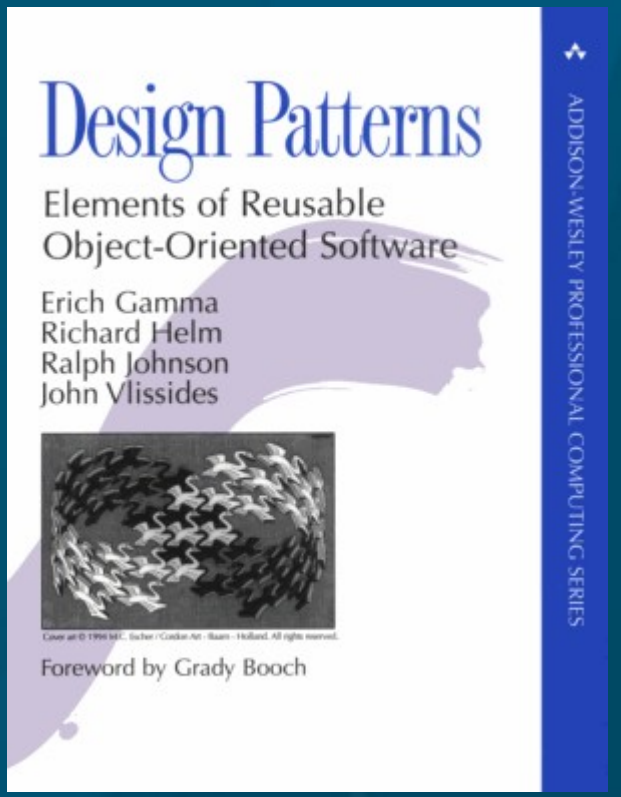
- Testing + coding is faster than just coding
- Ensures your code does what it's supposed to do
  - Ensures you know what it's supposed to do!
- Unit testing vs. integration testing
- Coding to a contract:
  - Contract → tests → design → code

- Make it run → make it right
  → make it fast → make it small

UML: Unified Modelling Language

- Developed at same time as OO
- Many kinds of diagrams
  - Class diagram
  - Inheritance hierarchy
  - Use-case scenarios
- Original idea from Christopher Alexander, “Notes on the Synthesis of Form”, Harvard University Press, 1964
- Ref: Gamma, Helm, Johnson, Vlissides, “Design Patterns: Elements of Reusable OO Software”

Design patterns



- A pattern is a named abstraction
  - from a recurring concrete form
  - that expresses the essence of
  - a proven general solution
- A pattern has three parts:
  - some recurring problem from the real world
  - the context of the problem (when to solve it)
  - the rule telling us how to solve it
- Describe a class of problems and how to solve

Parts of a design pattern

- Name: should be meaningful
- Problem: desired goal and obstacles
- Context: preconditions on problem
- Forces: relevant constraints, trade-offs, caveats
- Solution: structure, relationships, how-to
- Related patterns: codependencies, “see also”
- Known uses: example applications

Classes of patterns (high to low)

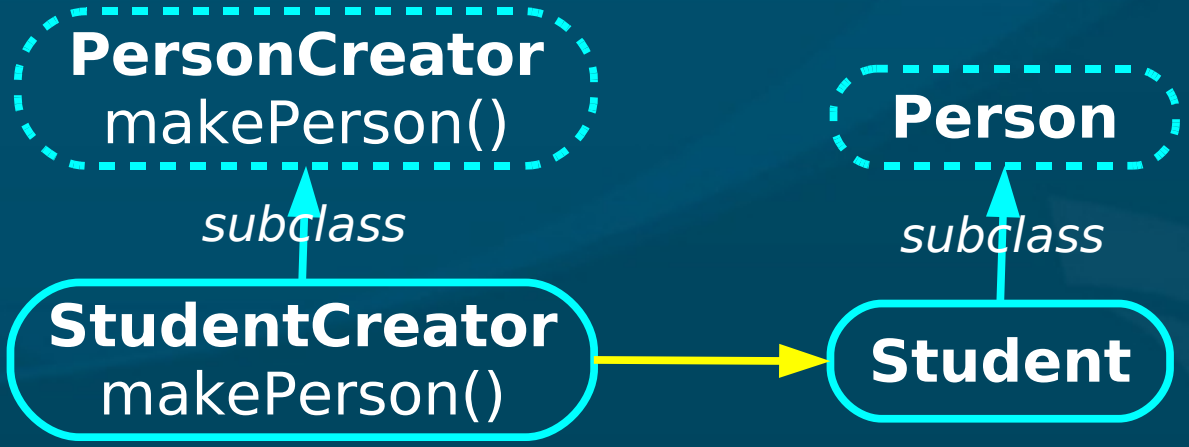
- Conceptual/architectural
  - Structural organization of software systems
  - Set of predefined components
  - Relationships between components
- Design
  - How to refine each component
  - Commonly recurring structure of components
- Programming idiom
  - How to code a particular component feature

Classes of patterns (GoF)

- Creational patterns
  - Interfaces to generate new objects
- Structural patterns
  - How to organize a large system in components
- Behavioural patterns
  - How components interact with each other to accomplish a common goal

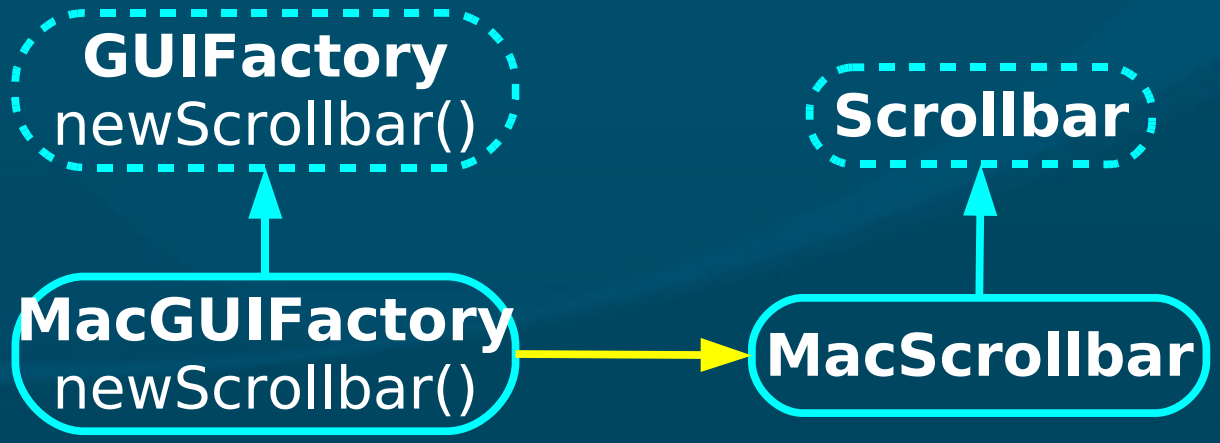
Creational: factory method

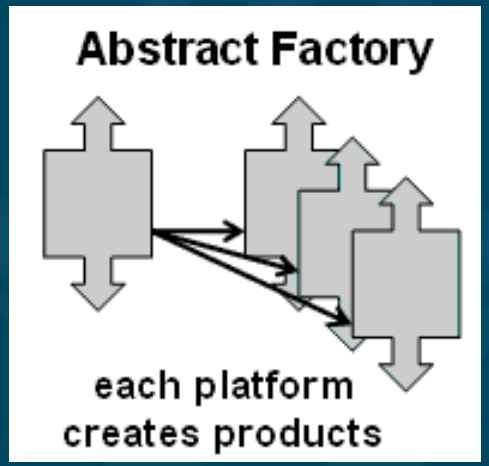
- Define an interface for creating an object, but let subclasses decide which class to instantiate
  - "Virtual constructor"
- Analogy: plastic injection-mould
- e.g., need to create a new Person; don't know in advance if it's Student, Staff, or Faculty



Creational: abstract factory

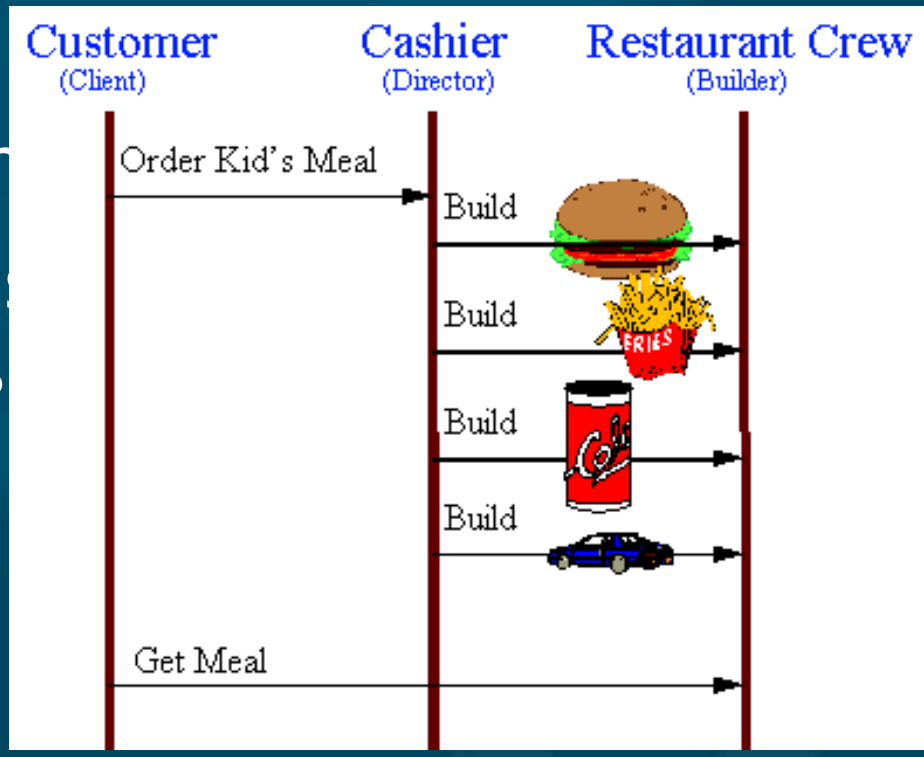
- Provide an interface to create families of related or dependent objects without specifying their concrete classes (“kit”)
  - e.g., adaptable look-and-feel of GUI widgets
- Analogy: press to stamp out auto parts
- May use a collection of factory methods

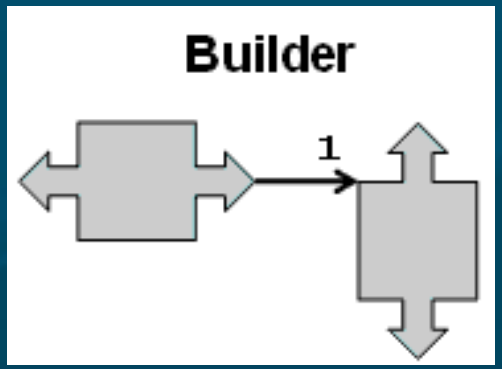




Creational pattern: builder

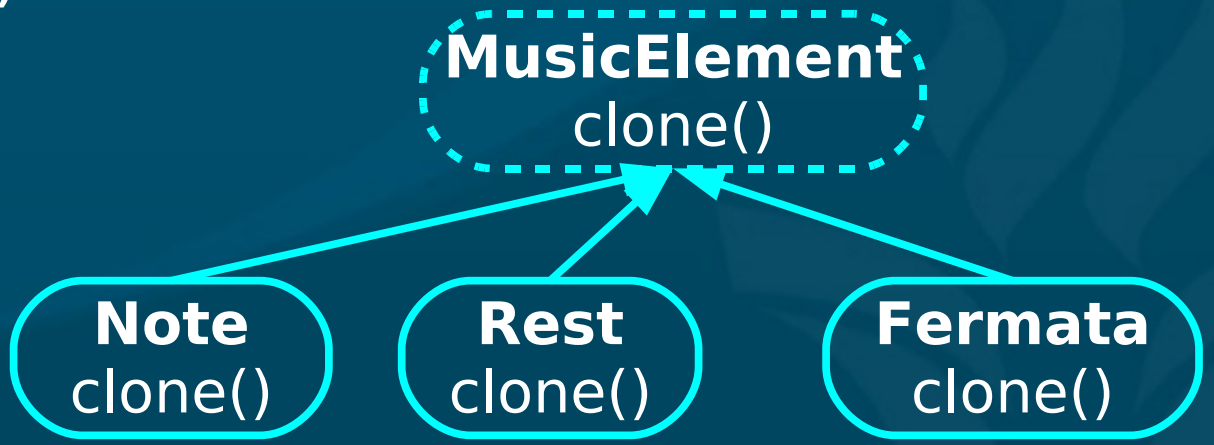
- Separate construction of a complex object from its representation
  - Analogy: assembling fast food kids' meals
- Director class parses the request and representation
- Hierarchy of Builder classes actually makes the objects





Creational pattern: prototype

- Create new objects by copying a prototype
  - Analogy: biological cell division
  - e.g., sheet-music editor: copy and paste notes
    - Staves are objects; each note is an object
  - Design each object so it knows how to copy itself: clone() method



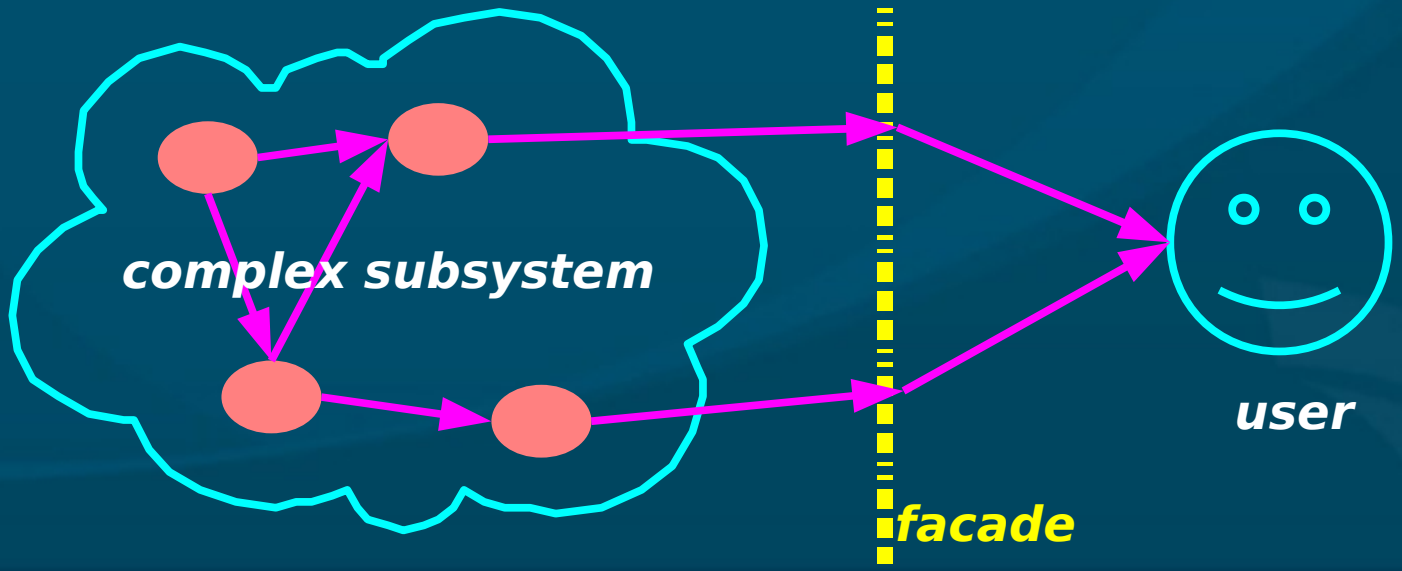
Creational pattern: singleton

- Ensure a class only has one instance, and provide a global point of access to it.
  - Analogy: only one President
- Often implement by making constructor private
  - Instantiate using static (class) method
    - Method checks if instance already exists
    - class President {

```
private:
  President thePrez;
  President() {}
public:
  static getPrez() { if (thePrez) return thePrez; }
```

Structural patterns: facade

- Provide a unified interface to a set of interfaces in a subsystem
  - High-level interface: system is easier to use
  - e.g., web front-end to complex database:
    - want minimal number of widgets, input boxes

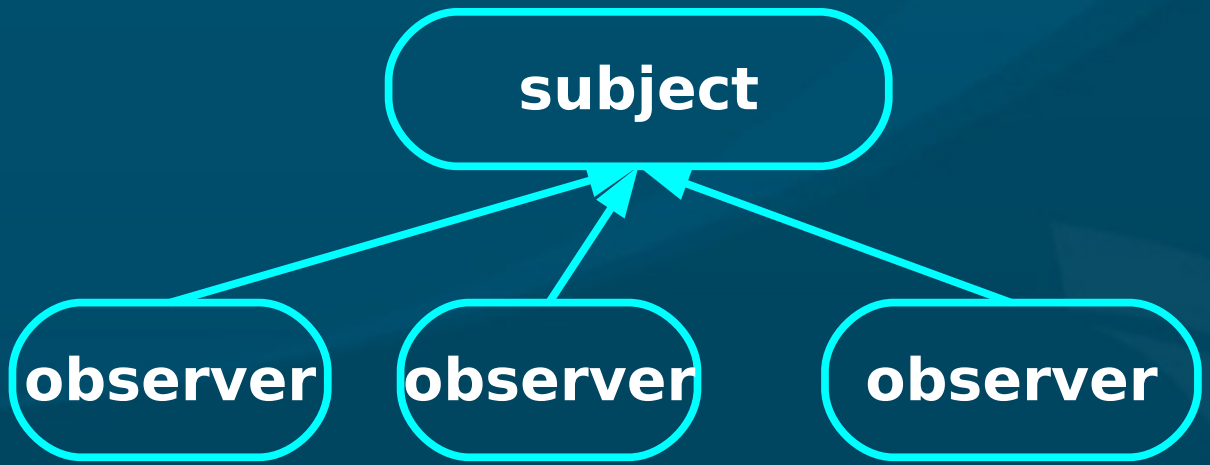


Other structural patterns

- Adapter/ wrapper: Convert the interface of a class into another interface clients expect
  - Lets otherwise incompatible classes cowork
- Bridge: decouple an abstraction from its implementation so they can vary independently
- Proxy: surrogate/placeholder for another object
- Decorator: dynamically add responsibilities / functionality to an object
- Flyweight: use sharing to support large numbers of fine-grained objects efficiently

Behavioural patterns: observer

- One-to-many dependency between objects so that when the subject changes state, all its observers are notified and updated
  - e.g., many students checking TWU website for snow closures
  - e.g., server message "send to all" clients



Other behavioural patterns

- Mediator: an object that encapsulates how a set of other objects interact.
  - Promotes loose coupling by keeping objects from referring to each other directly
- Chain of responsibility: avoid coupling sender directly to receiver by passing through chain
- Iterator: access all elements of a collection
- Memento: save/restore state of an object
- Command: make requests objects
  - queue/log requests, support undo, etc.