Design Patterns (1)

7 Apr 2010
CMPT166
Dr. Sean Ho
Trinity Western University

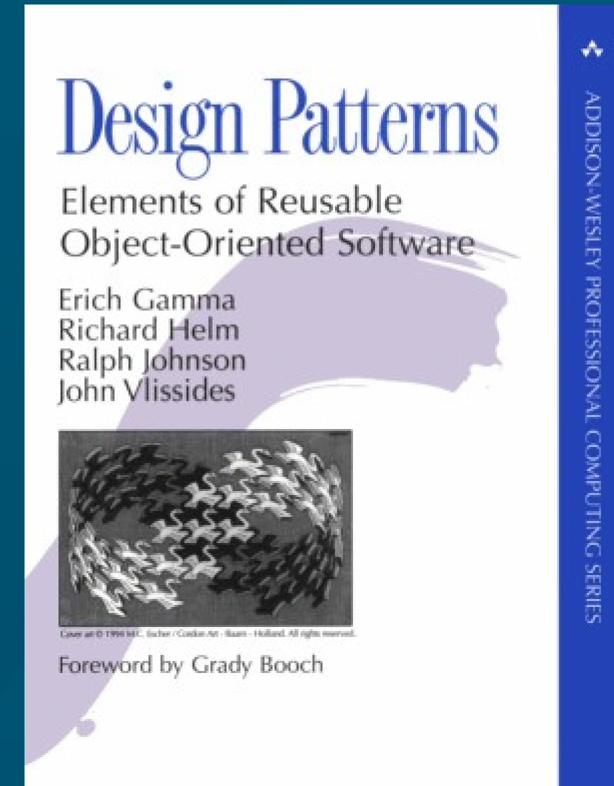
See also:
Vince Huston,
JavaCamp

TRINITY WESTERN UNIVERSITY

UML and reusable designs

- Diagrams for
  - Use-case scenarios
  - Component / CRC diagrams
  - Class diagram
  - Sequence diagram
- 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: co-dependencies, “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

The Sacred Elements of the Faith

the holy origins

the holy behaviors

the holy structures

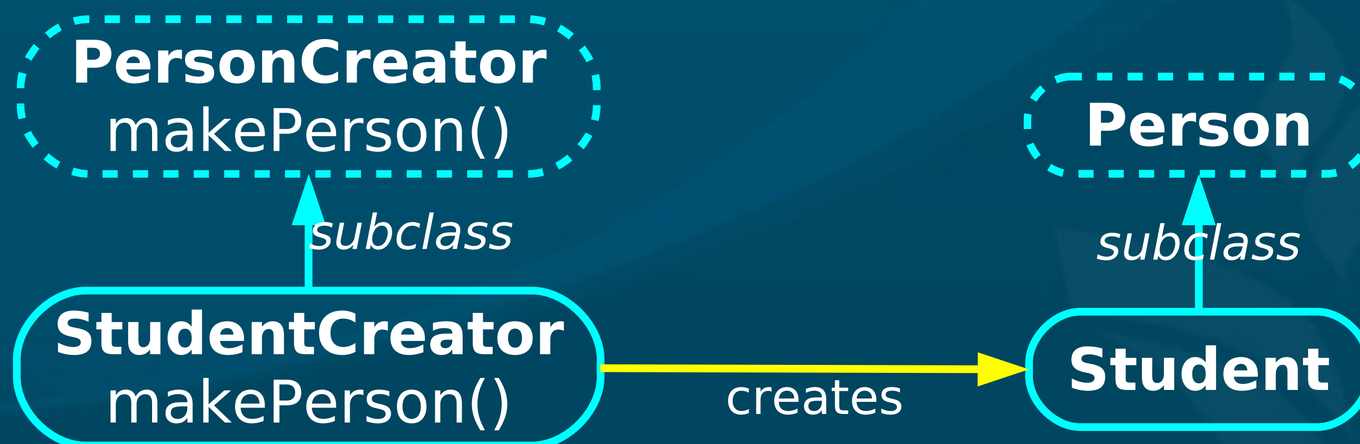
107 FM Factory Method							139 A Adapter
117 PT Prototype	127 S Singleton				223 CR Chain of Responsibility	163 CP Composite	175 D Decorator
87 AF Abstract Factory	325 TM Template Method	233 CD Command	273 MD Mediator	293 O Observer	243 IN Interpreter	207 PX Proxy	185 FA Façade
97 BU Builder	315 SR Strategy	283 MM Memento	305 ST State	257 IT Iterator	331 V Visitor	195 FL Flyweight	151 BR Bridge

Creational patterns

- Factory Method: create a variety of objects
- Abstract Factory: group of related obj factories
- Builder: delegate creation of components
- Prototype: clone a template object
- Singleton: enforce having only one instance

Creational: factory method

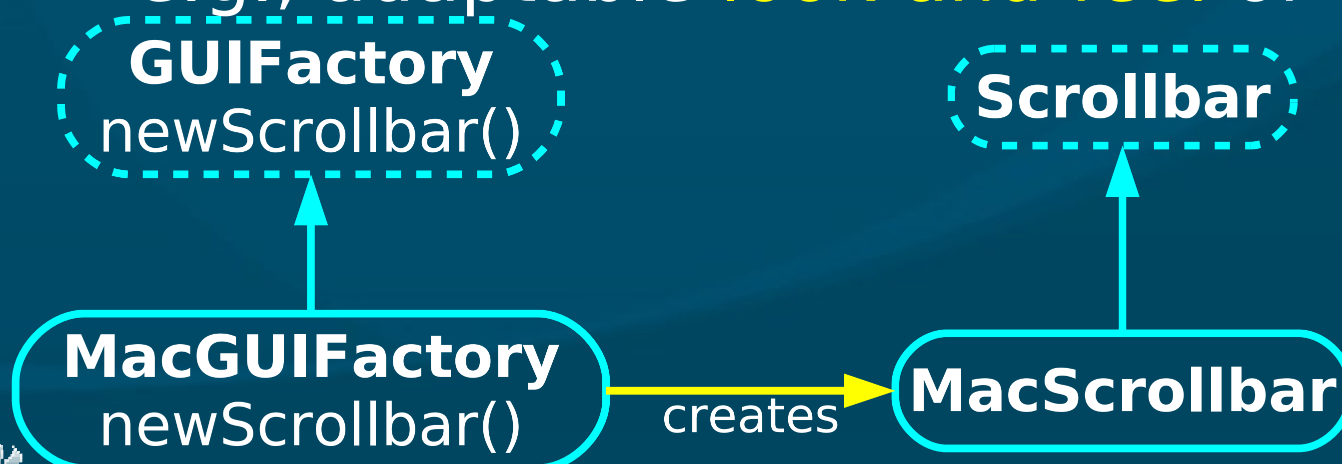
- An interface to create an object, but without specifying which subclass
- Analogy: plastic injection-mould determines shape of output
- e.g., need to create a new Person; don't know in advance if it's Student, Staff, or Faculty

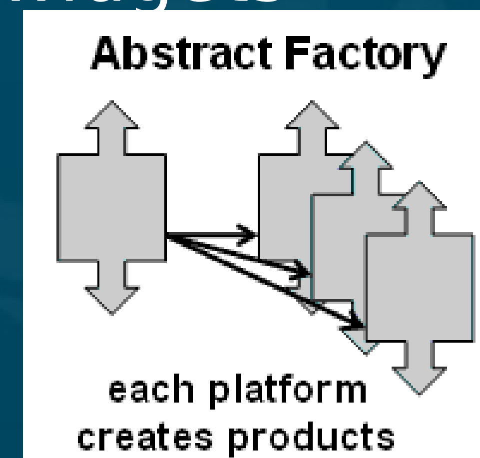


Creational: abstract factory

JavaCamp, Wikipedia

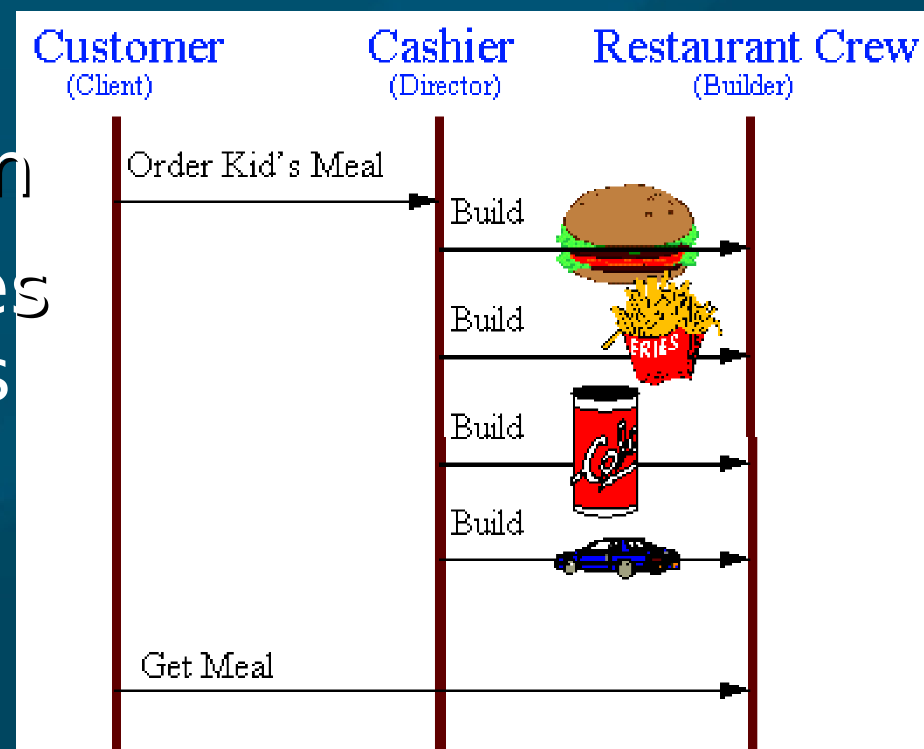
- Family of similar factories
  - Client code doesn't know/care which concrete factory is used
  - May use a collection of factory methods
- Analogy: press to stamp out auto parts
- e.g., adaptable look-and-feel of GUI widgets





Creational pattern: builder

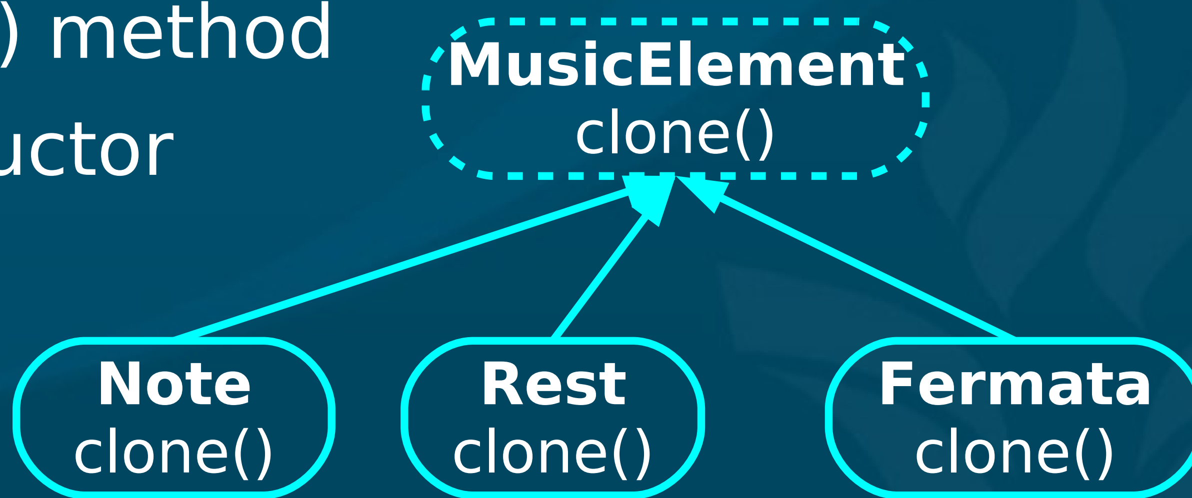
- Separate construction of a complex object from its representation
  - Analogy: assembling fast food kids' meals
- Director class parses the request and representation
- Hierarchy of Builder classes actually makes the objects



Creational pattern: prototype

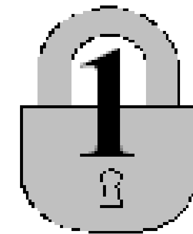
JavaCamp, Wikipedia

- 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
  - Copy constructor



JavaCamp, Wikipedia

Creational pattern: singleton



- Ensure a class only has one instance, and provide a global point of access to it
    - Analogy: only one Prime Minister
- Often implement by making constructor private
    - Provide a static get method for the singleton
        - public class PrimeMinister {

```
  private PrimeMinister thePM;
  private PrimeMinister() { /* create new PM */ };
  public static getPM() {
    if (!thePM) thePM = new PrimeMinister();
    return thePM; }
```

Structural patterns

- Facade: unified/simplified interface to system
- 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

Structural pattern: 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