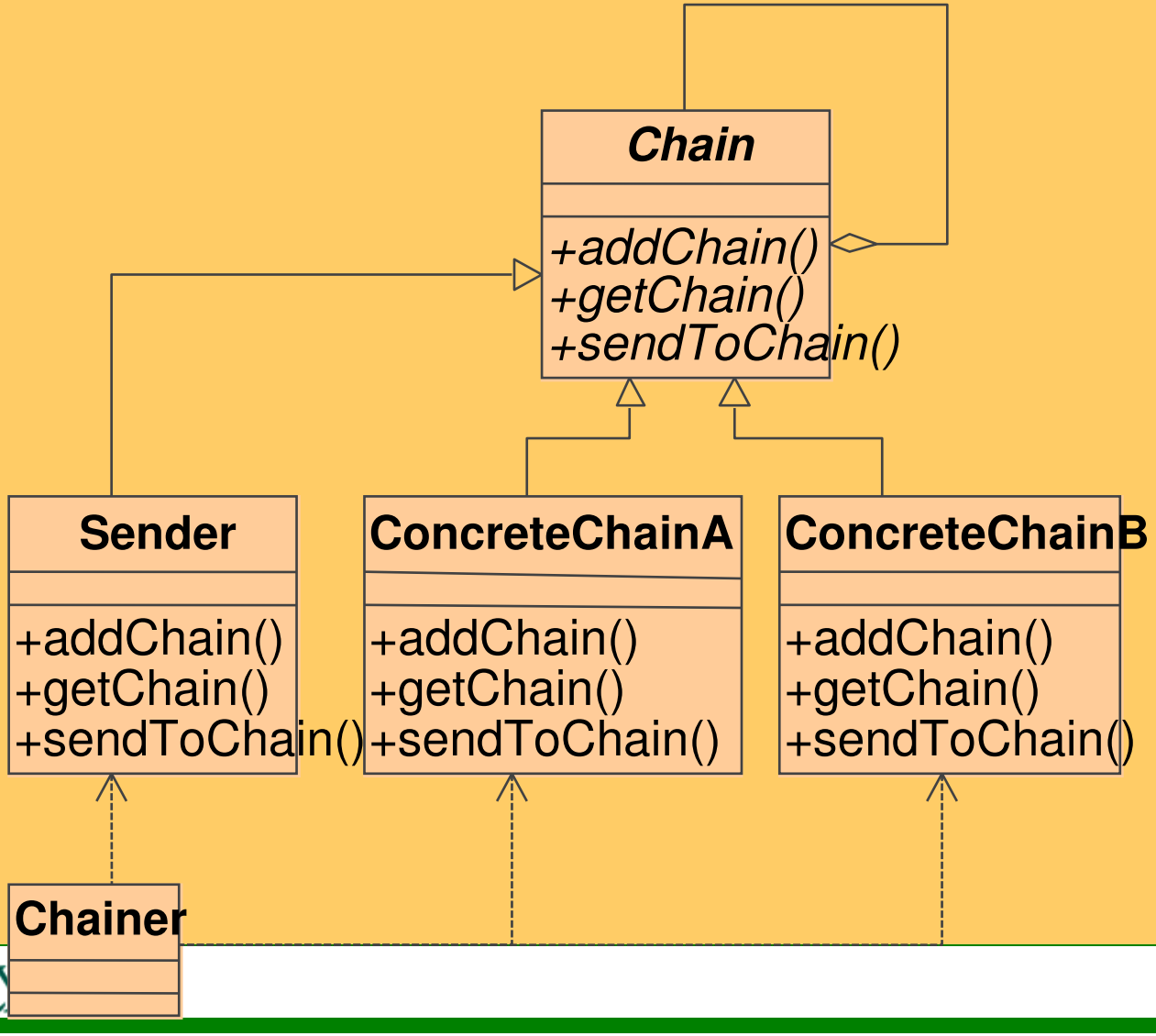
Design Patterns

David Janzen

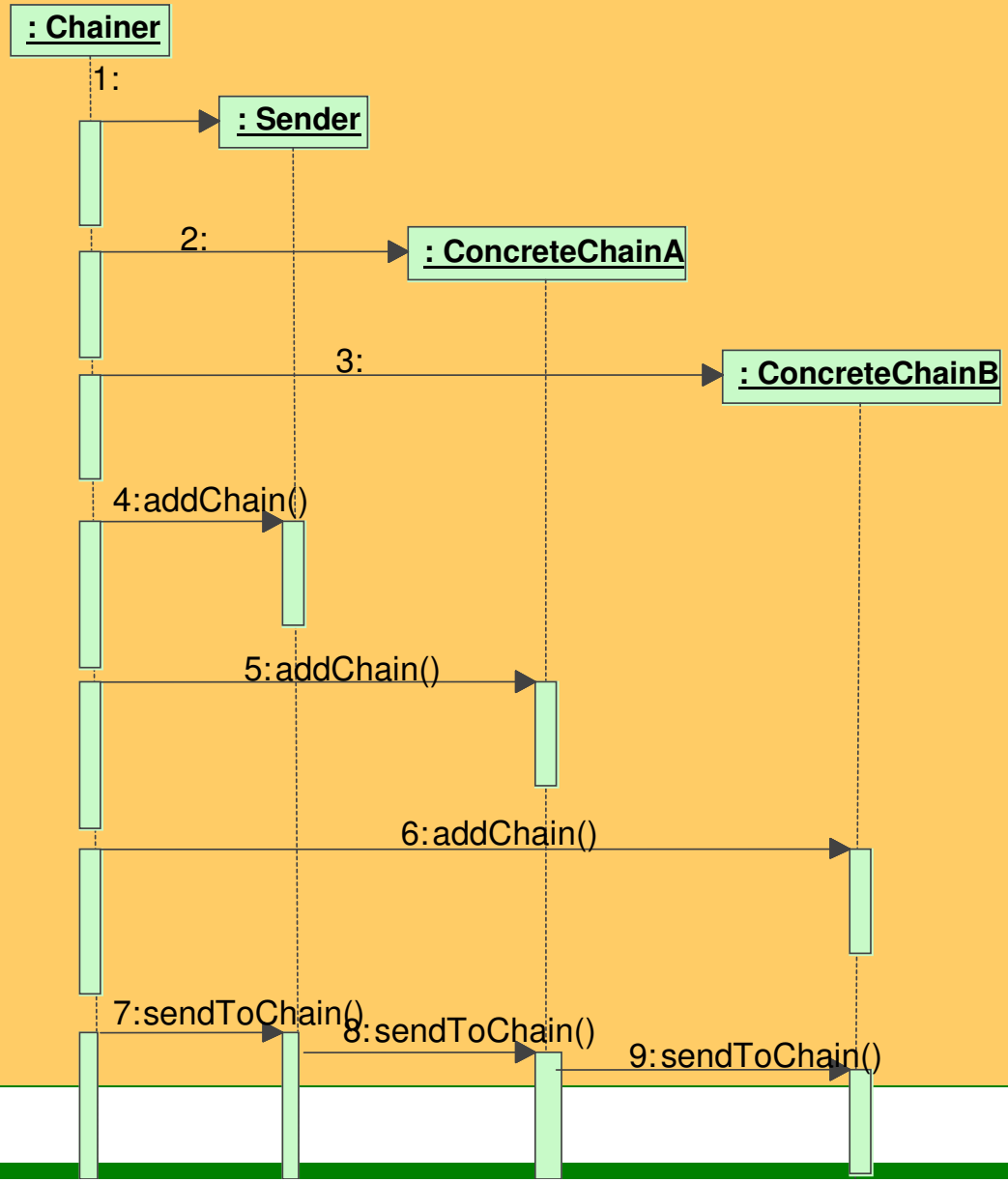
Chain of Responsibility Pattern

- Intent: (p. 223)
  - Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.
- Basic Idea:
  - Decouple a request sender from its receiver by allowing multiple objects the chance to handle a request. Each object only knows the next object in chain
- Game:
  - variation on hot potato
  - always pass objects in same order
  - each person knows what kind of object to keep
    - e.g. candyeater eats candy, jackspinner, ballbouncer,…

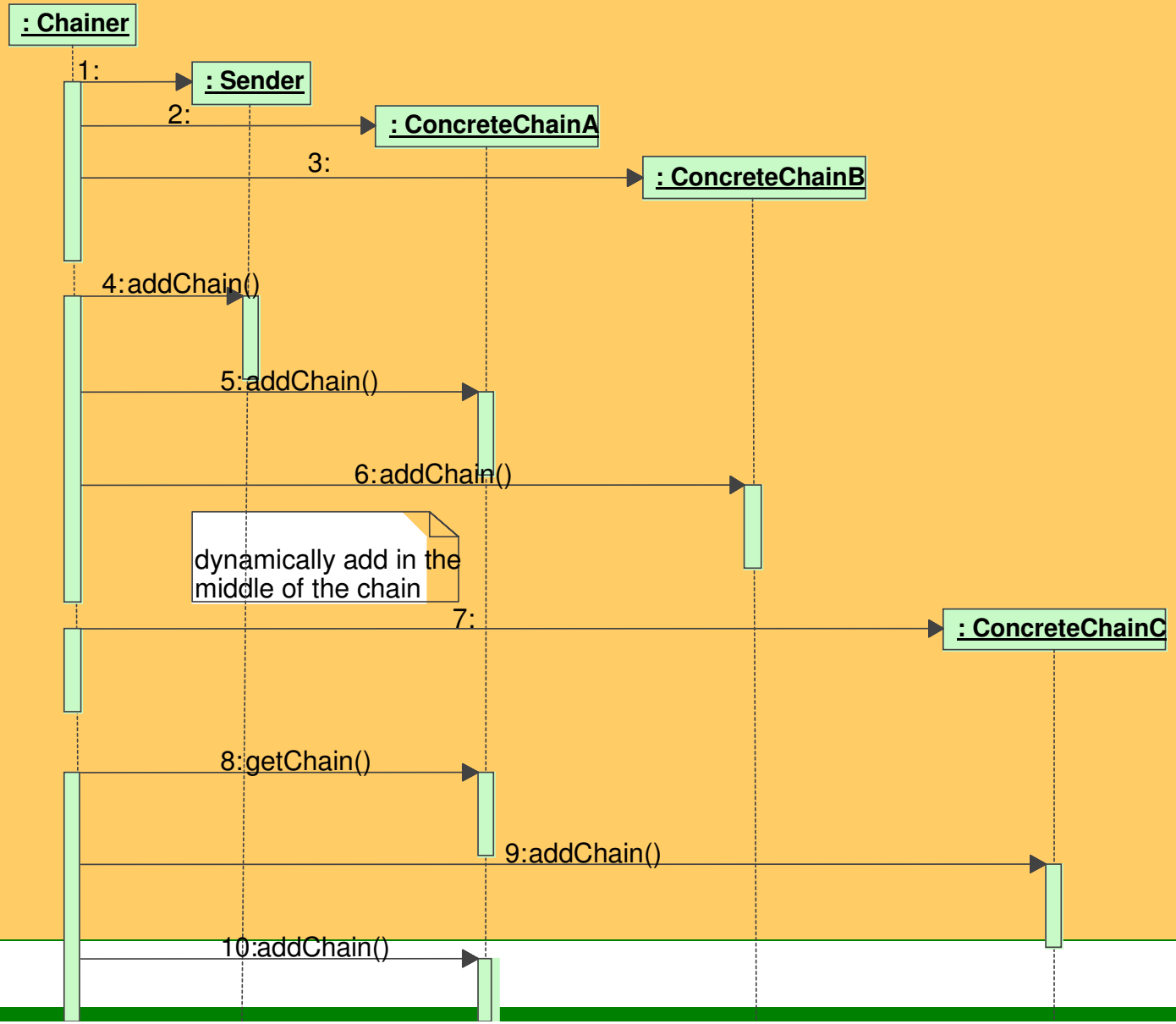
Chain of Responsibility Pattern
General Structure



Chain of Responsibility Pattern

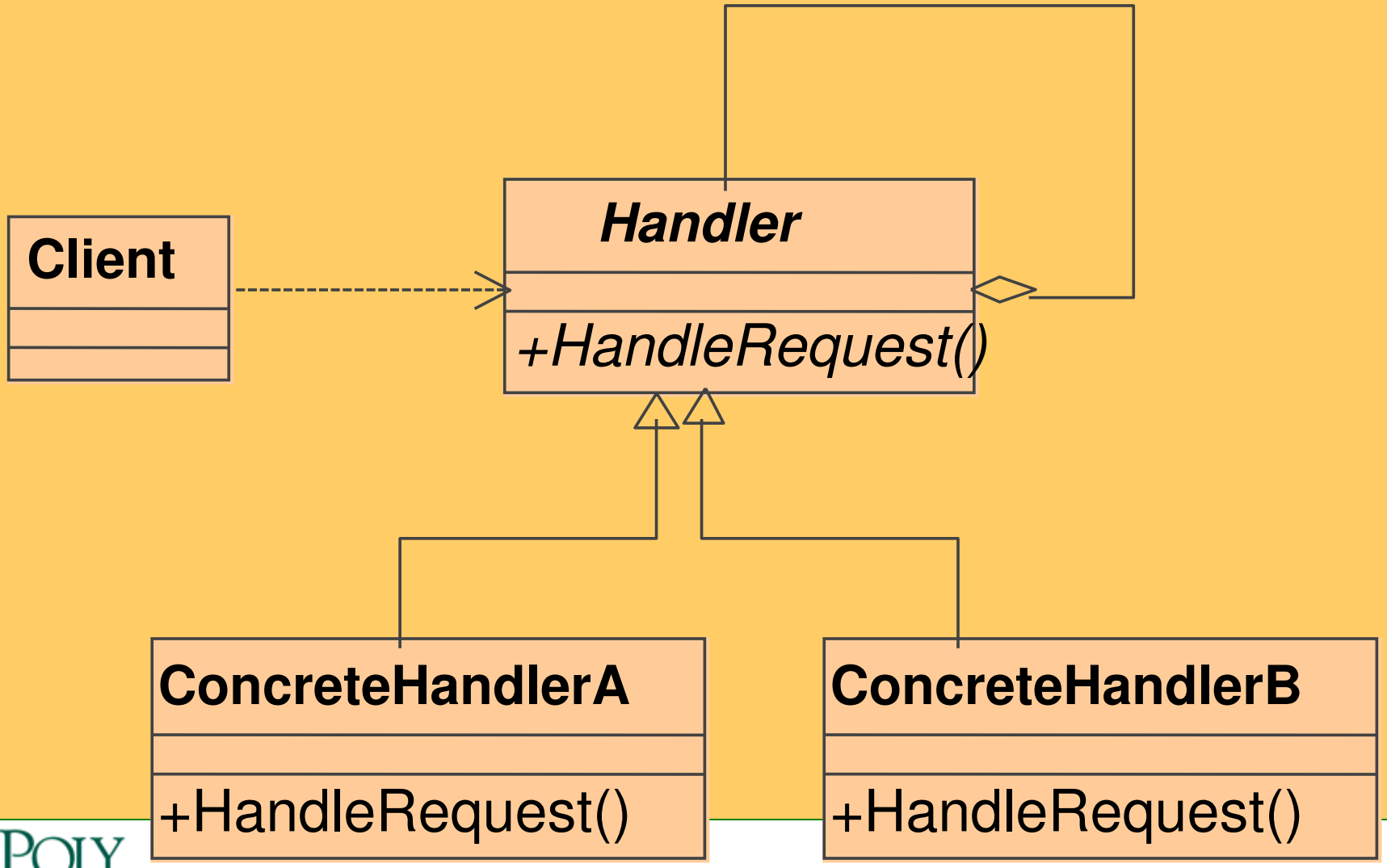


Chain of Responsibility Pattern



Chain of Responsibility Pattern

General Structure



Chain of Responsibility Pattern

- Handler:
  - Chain (defines interface for handling requests and passing them on to the next object in the chain if not handled)
- ConcreteHandler:
  - Sender, Imager, FileList, RestList (handles requests or sends request to next object in the chain)
- Client:
  - Chainer, Sender (sets up the chain and submits the request to be handled to the first object in the chain)
- Request:
  - Optionally might create an abstract request type for more complex requests

Chain of Responsibility Pattern

```
public interface Chain
{
public abstract void addChain(Chain c);
       //make this class “point” to c
public abstract void sendToChain(String mesg);
       //send request to next object in chain
public Chain getChain();
       //return the next object in the chain
}
```

Example from “Java Design Patterns: A Tutorial” by James A. Cooper

Chain of Responsibility Pattern

```
public class Imager implements Chain {
    private Chain nextChain;
//---------------------------------------
    public void addChain(Chain c) {
        nextChain = c;    //next in chain of resp
    }
//---------------------------------------
    public void sendToChain(String mesg) {
        //if there is a JPEG file with this root name
        //load it and display it.
        if (findImage(mesg))
            loadImage(mesg+".jpg");
        else
            //Otherwise, pass request along chain
            nextChain.sendToChain(mesg);
    }
//---------------------------------------
    public Chain getChain() {
        return nextChain;
    }
```

Chain of Responsibility Pattern

```
public class Chainer
{
   //list of chain members
   Sender sender;        //gets commands
   Imager imager;        //displays images
   FileList fileList;    //highlights file names
   ColorImage colorImage;  //shows colors
   RestList restList;    //shows rest of list
 public Chainer() {
   sender = new Sender();
   imager = new Imager();      //add all these to the Frame
   fileList = new FileList();
   colorImage = new ColorImage();
   restList = new RestList();
          //set up the chain of responsibility
   sender.addChain(imager);
   imager.addChain(colorImage);
   colorImage.addChain(fileList);
   fileList.addChain(restList);
```

Chain of Responsibility Pattern

- When to use:
  - When more than one object may handle a request and we don't know the handler beforehand
  - When we want to decouple a request sender from its request receivers
  - When we want to dynamically change the objects that might handle a request
- Consequences
  - A request might not get handled
  - Reduced coupling; neither sender nor receiver know of the other

What are Design Patterns?

- In its simplest form, a pattern is

  a solution to a recurring problem

    in a given context

- Patterns are not created, but discovered or identified

Design Patterns Definition[1]

- Each pattern is a three-part rule, which expresses a relation between
  - a certain context,
  - a certain system of forces which occurs repeatedly in that context, and
  - a certain software configuration which allows these forces to resolve themselves

[1] Dick Gabriel, http://hillside.net/patterns/definition.html

A Good Pattern[1]

- *Solves a problem*:
  - Patterns capture solutions, not just abstract principles or strategies.
- *Is a proven concept*:
  - Patterns capture solutions with a track record, not theories or speculation

[1] James O. Coplien, http://hillside.net/patterns/definition.html

A Good Pattern

- *The solution isn't obvious*:
  - Many problem-solving techniques (such as software design paradigms or methods) try to derive solutions from first principles. The best patterns *generate* a solution to a problem indirectly--a necessary approach for the most difficult problems of design.
- *It describes a relationship*:
  - Patterns don't just describe modules, but describe deeper system structures and mechanisms.

A Good Pattern

- *The pattern has a significant human component (minimize human intervention).*
  - All software serves human comfort or quality of life; the best patterns explicitly appeal to aesthetics and utility.

Why Patterns?

- Code Reuse is good
  - Software developers generally recognize the value of reusing code
    - reduces maintenance
    - reduces defect rate (if reusing good code)
    - reduces development time
  - Code Reuse Pitfalls
    - Reusing code blindly (out of context)
    - Reusing unproven or untested code

Why Patterns?

- Design Reuse may be even better
  - Similar benefits and pitfalls to Code Reuse
  - Identifying reuse early in the development process can save even more time
  - Copying proven solutions
  - Solutions can be analyzed visually with UML without complexities of the code

Why Patterns?

- Good designs reduce dependencies/coupling
  - Many patterns focus on reducing dependencies/coupling
  - Strongly coupled code
    - hard to reuse
    - changes have wide effects
  - Loosely coupled code
    - objects can be reused
    - changes have isolated effects

Why Patterns?*

- Common vocabulary and language
  - Fundamental to any science or engineering discipline is a common vocabulary for expressing its concepts, and a language for relating them together.
  - Patterns help create a shared language for communicating insight and experience about recurring problems and their solutions.

* http://hillside.net/patterns

Why Patterns?

- Body of solutions literature
  - The goal of patterns within the software community is to create a body of literature to help software developers resolve recurring problems encountered throughout all of software development.

Why Patterns?

- Encapsulation enables higher reasoning
  - Forming a common pattern language for conveying the structures and mechanisms of our architectures allows us to intelligibly reason about them.

Why Patterns?

- Abstract the technology
  - The primary focus is not so much on technology as it is on creating a culture to document and support sound engineering architecture and design.

Design Patterns History

- Christopher Alexander (architect)

  "A Pattern Language", 1977

  – quality architectural designs can be
    - described
    - reused
    - objectively agreed to be good
  – structures solve problems
  – multiple structures can solve the same problem
  – similarities in these structures can form a pattern

* see www.greatbuildings.com/architects/Christopher_Alexander.html and http://c2.com/cgi/wiki?ChristopherAlexander

Design Patterns History

- ESPRIT consortium in late 1980's developed a pattern-based design methodology inspired by Alexander's work
- OOPSLA'87 Kent Beck and Ward Cunningham introduced idea of identifying patterns in software engineering
  - http://c2.com/cgi/wiki?WardAndKent
  - Well known for work in Smalltalk, CRC Cards, xUnit testing framework, eXtreme Programming, …

Design Patterns Resources

- "Design Patterns: Elements of Reusable Object-Oriented Software"
  - by Gamma, Helm, Johnson, and Vlissides
  - Gang of Four (GoF)
  - 1994 Software Productivity Award
- http://hillside.net/patterns/patterns.html
- http://patterndigest.com
- http://wiki.cs.uiuc.edu/PatternStories/

Design Patterns

- GoF grouped patterns into three areas:
  - Creational Patterns
    - Abstract Factory, Builder, Factory Method, Prototype, Singleton
  - Structural Patterns
    - Adapter, Bridge, Composite, Decorator, Façade, Flyweight, Proxy
  - Behavioral Patterns
    - Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method, Visitor

Design Patterns

- GoF book preceded commercial release of Java
- Java language designers read GoF
- Hence Java libraries implement many GoF Design Patterns
  - Iterator on Vector or ArrayList
  - Observer (interface) and Observable (class)
  - WindowAdapter
  - Stack

Creational Patterns

- Better ways to create an instance
- Abstract the instantiation process
  - hide how class instances are created and combined
  - Traditional creation:
    - MyType t = new MyType();
      - client code is now dependent on MyType
      - changing MyType means changing all clients
      - clients must know of different ways to create MyType

Structural Patterns

- Ways to combine classes and objects into larger structures
- Structural *Class* Patterns use inheritance to compose interfaces or implementations
  - Multiple Inheritance, Adapter
- Structural *Object* Patterns compose objects to realize new functionality
  - Adapter, Bridge, Composite, Decorator, Façade, Flyweight, Proxy

Behavioral Patterns

- Ways to deal with algorithms and have objects communicate
- Behavioral *Class* Patterns use inheritance to distribute behavior between classes
  - Template Method, Interpreter
- Behavioral *Object* Patterns compose objects to distribute behavior
  - Mediator, Chain of Responsibility, Observer, Strategy, Command, State, Visitor, Iterator, Memento

Ask what changes?

- if algorithm changes: use Strategy
- if only parts of an algorithm change: use Template Method
- if interaction changes: use Mediator
- if subsystem changes: use Façade
- if interface changes: use Adapter
- if number and type of dependents changes: use Observer
- if state causes behavior changes: use State
- if object to be instantiated changes: use Factory
- if the action an object is requested to perform changes: use Command

Patterns and Frameworks

- "A framework is an integrated set of components that collaborate to provide a commoditized software architecture for a family of related applications. Mature frameworks exhibit high pattern density, making patterns an ideal descriptive tool for developing, evolving, and understanding frameworks."

from "Past, Present, and Future Trends in Software Patterns," Buschman et al.