GE Java Sessions

Introduction to Software Patterns

Mohamed Wiem Mkaouer

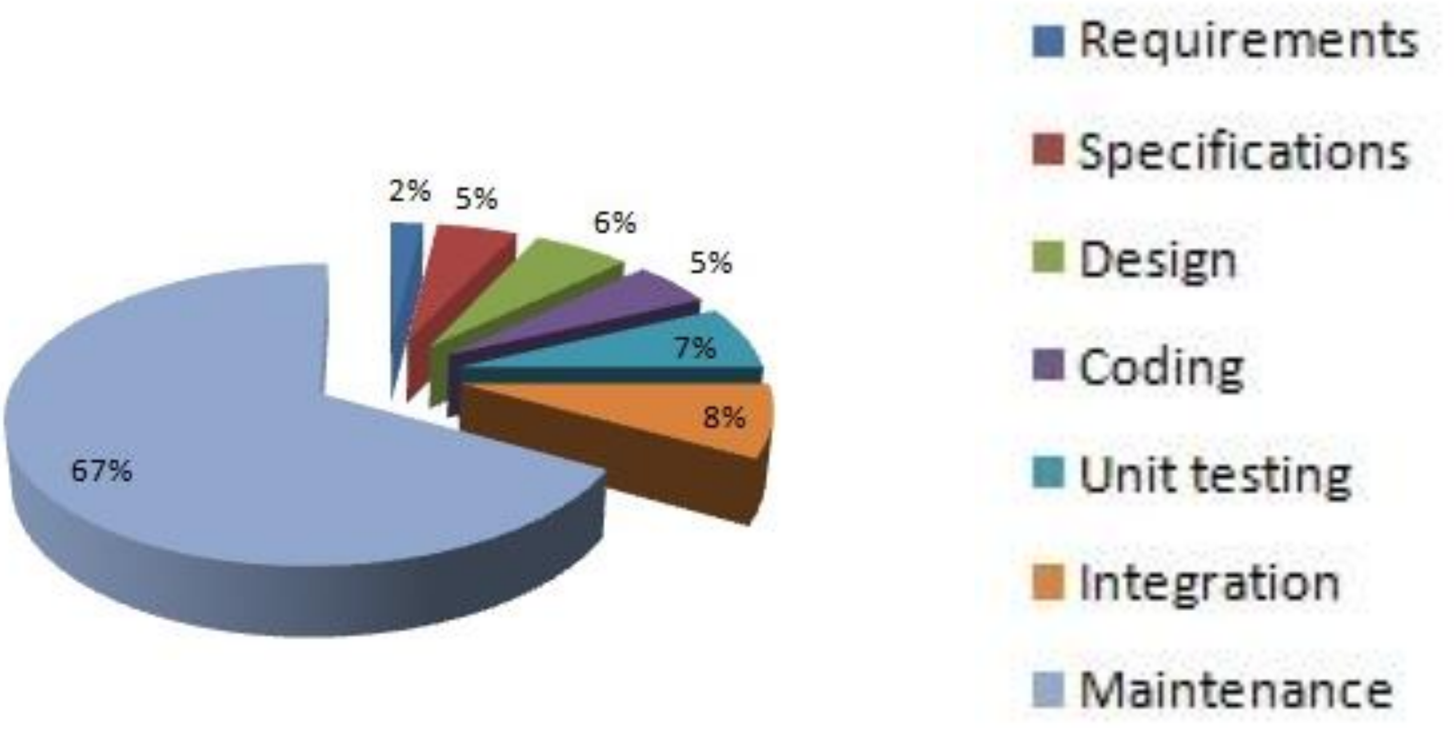
mmkaouer@umich.edu

Software Architecture and Reusability

- Software architecture
  - Consists of software components, their external properties, and their relationships with one another.
  - It also refers to documentation of a system's software architecture.
- Developing software is hard
- Developing reusable software is even harder
  - Reusability
    - Likelihood that a module can be used again to add new functionalities with slight or no modification.
- Software Patterns provide proven solution
  - Reusable elements

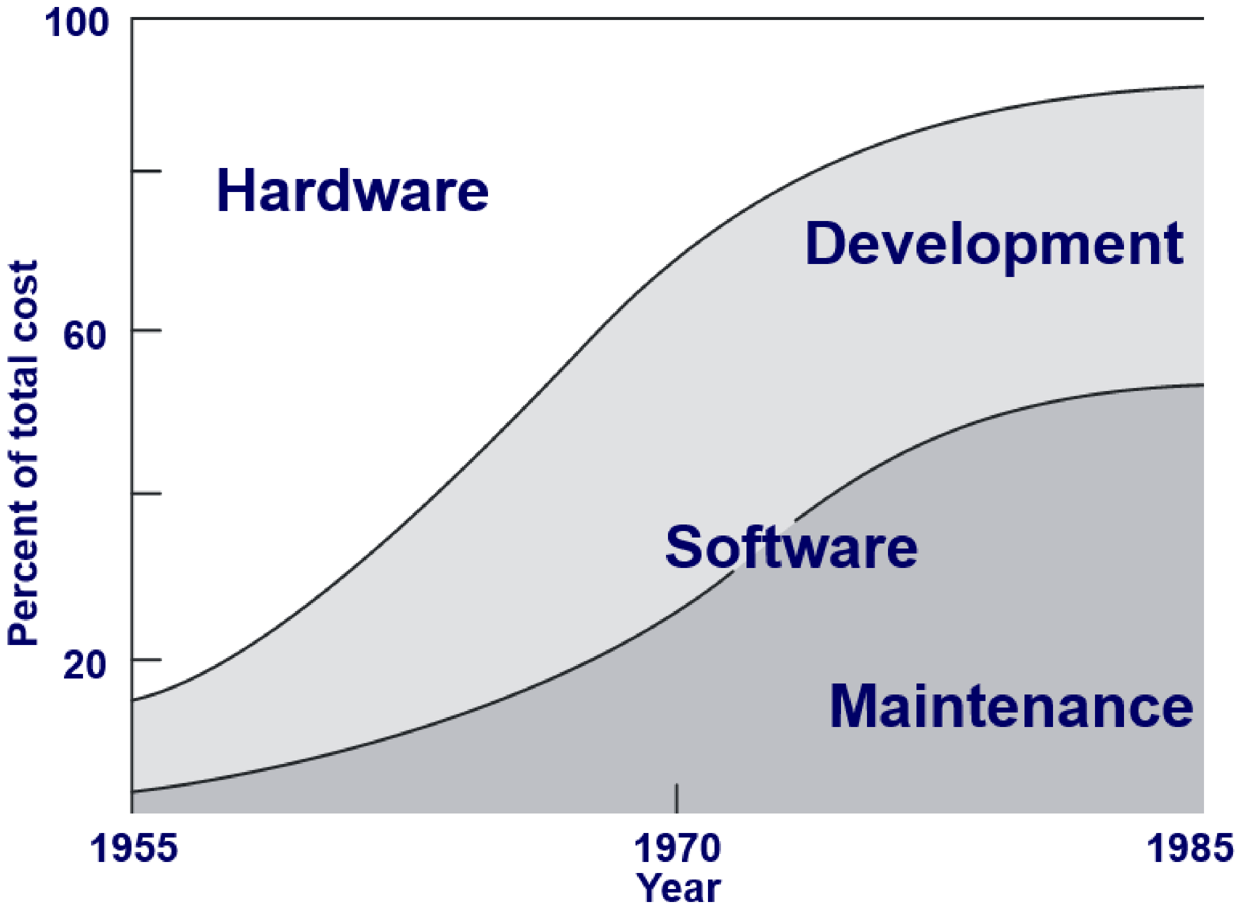
Why Design patterns?

- Software Life-Cycle Costs



Why Design patterns?

- Software Maintenance Evolution

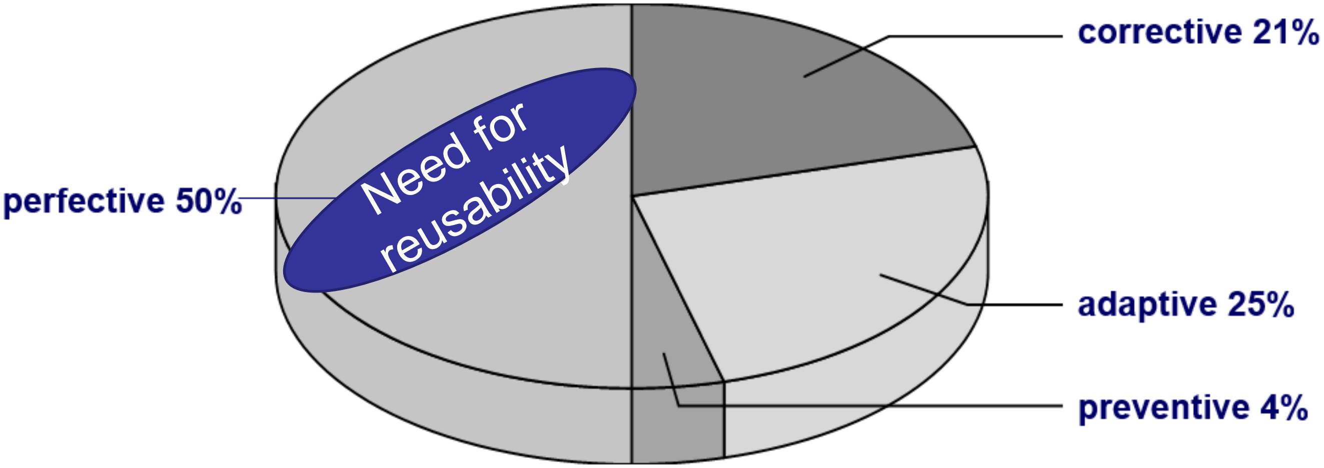


Why Design patterns?

- Software Maintenance Evolution

Why Design patterns?

- Software Maintenance Categories
  1. **Perfective maintenance:** changes required as a result of user requests (a.k.a. *evolutive* maintenance)
  2. **Adaptive maintenance:** changes needed as a consequence of operated system, hardware, or DBMS changes
  3. **Corrective maintenance:** the identification and removal of faults in the software
  4. **Preventative maintenance:** changes made to software to make it more maintainable



Patterns in the Webster Dictionary

- A form or model proposed for imitation
- Something designed or used as a model for making things (e.g., a dressmaker's pattern)
- …

Learning from Experts

- When experts work on a problem:
  - They typically do not invent a new solution.
  - They know from their own experience and the experience of other people a set of design solutions.

- If they face a new problem:
  - They often remember how they solved similar problem and adopt an old solution to a new context

Experts think in problem/solution pair

Becoming a Software Design Master

- First learn the rules
  - Algorithms, data structures, etc.
- Then learn the principles
  - Structured programming, modular programming, OOAD etc.
- However, to truly master software design, one must study designs of other masters
  - These designs contains patterns that must be understood, memorized and applied repeatedly in context.
- There are hundreds of such software design patterns

Design Patterns

- Represent solutions to problems that arise when developing software
  - “Pattern = problem/solution pair applied in context”
- Capture the static and dynamic structure and collaboration among key participants in software designs
- Facilitate reuse of successful software architectures and designs

Origins of Design Patterns

- 1988-1991: Erich Gamma, Ph. D. thesis
- 1989 -1991: James Coplien, Advanced C++ Idioms book
- 1994-present: PLoP Conferences and books
- 1995: Group of Four (GoF) - “Design Pattern:Elements of Reusable OO software”
- 1996: Buschmann, Meunier, Rohnert, Sommerland, Stal - Pattern-Oriented Software Architecture: A System of Patterns (“POSA book”)

Origins of Design Patterns (cont’d)

- Nowadays
  - Many reports and published articles support the benefit of use of patterns
  - ACM software engineering curriculum has included software design pattern topic

Design Patterns: Definition I

- A general repeatable solution to a commonly occurring problem in software design.
- It is not a finished design that can be transformed directly into code.
- It is a description or template for how to solve a problem that can be used in many different situations.
- Object-oriented design patterns typically show relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved.

Design Patterns: Definition II

- GoF
  - A design pattern is a description of communicating objects and classes that are customized to solve a general design problem in a particular context

Design Pattern: Example

- A system that uses a number of temperature sensors to monitor the condition of a hardware device.
- The system uses specific sensors: TempTek, Inc. TS7000 sensors.
- TempTek supplies a simple Java class to interface with the sensors:

```
Class TS7000 {
    native double getTemp();
    ...
}
```

Inspired From William H. Mitchell’s presentation

Design Pattern: Example (cont’d)

- Monitoring code that simply calculates the mean temperature reported by the sensors.

```
double sum = 0.0;
for (int i = 0; i < sensors.length; i++)
sum += sensors[i].getTemp();
double meanTemp = sum / sensors.length;
```

- Sensors is declared as an array of TS7000 objects.
  - (TS7000 sensors[ ] = new TS7000[...])

Design Pattern: Example (cont’d)

- Assume now that the system uses a mix of TS7000s and sensors from a new vendor, Thermon.
- The Thermon sensors are SuperTemps and a hardware interfacing class is supplied:

```
Class SuperTempReader {
// NOTE: temperature is Celsius tenths of a degree
native double current_reading();
...
}
```

Design Pattern: Example (cont’d)

- Here is the monitoring code:

```
For (int i = 0; i < sensors.length; i++)
{
If (sensors[i] instanceof TS7000)
sum += ((TS7000)sensors[i]).getTemp();
Else
// Must be a SuperTemp!
sum +=
((SuperTempReader)sensors[i]).current_reading() * 10;
}
```

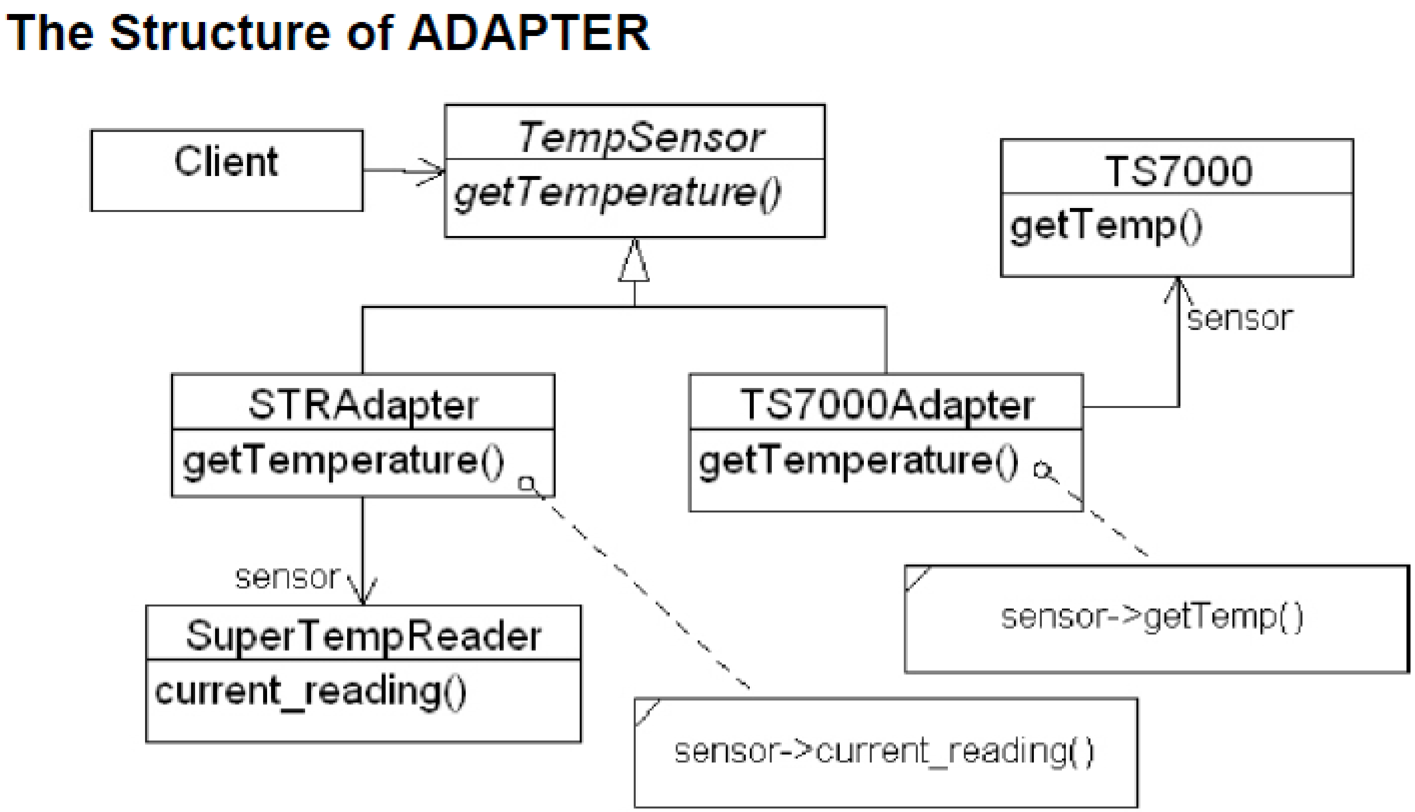
- Sensors is an array of Objects.
  - The type is tested with instanceof and an appropriate cast and method call is performed.

A Pattern to the Rescue

- Problems arose when a component from a second vendor was introduced.
  - More vendors may be involved in the future.
- We have no control over the name of the temperature-reporting method in the vendor-supplied classes.
- The value produced may need scaling, unit conversion, etc.
- All that we can really expect is that a temperature can be read from each sensor.

The Adapter Patter Provides a Solution to This Problem

Rescue Pattern: Adapter



The Adapter Pattern Provides a Solution to This Problem

What Makes It A Pattern?

- A Pattern must:
  - Solve a problem and be useful
  - Have a context and can describe where the solution can be used
  - Recur in relevant situations
  - Provide sufficient understanding to tailor the solution
  - Have a name and be referenced consistently

Benefits of Design Patterns

- Developers can have some confidence that the solution chosen is not entirely off the wall and has been used with success in similar situations in other systems.
- Patterns enable large-scale reuse of software architectures and also help document systems
- Patterns explicitly capture expert knowledge and design tradeoffs and make it more widely available
- Patterns help improve developer communication

But …

- Patterns are not a *panacea*
- Patterns are validated by experience and discussion rather than by automated testing
- Integrating patterns into a software development process is a human-intensive activity.

Anti-Patterns

- Also referred to as pitfalls,
- Classes of commonly-reinvented bad solutions to problems.
- They are studied, as a category, in order that they may be avoided in the future, and that instances of them may be recognized when investigating non-working systems.

Pattern Categories

- Architectural Patterns
- Design Patterns
- Idioms

Architectural Pattern

- Expresses a fundamental structural organization or schema for software systems.
- Provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them.

Design Pattern

- Provides a scheme for refining the subsystems or components of a software system, or the relationships between them.
- Describes commonly recurring structure of communicating components that solves a general design problem within a particular context.

Idiom

- Low-level pattern specific to a programming language.
- It describes how to implement particular aspects of components or the relationships between them using the features of the given language.

Describing Design Patterns in GoF

- Pattern name and classification
  - Contains the essence of pattern succinctly
  - Become part of your design vocabulary
- Intent
  - What does the pattern do ?
  - What particular problem does it address ?
- AKA: Other well-known names

GoF Description (cont’d)

- Motivation
  - Illustrate a design problem and how the class and the object structures solve the problem
- Applicability
  - In which situations the pattern can be applied?
  - How can you recognize these situations?

GoF Description (cont’d)

- Structure
  - Graphical representation of the classes and their collaborations in the pattern
- Participants
  - Class
  - Objects
  - Responsibilities

GoF Description (cont’d)

- Collaborations
  - How the participants collaborate to carry out their responsibilities
- Consequences
  - How does the pattern support its objectives?
  - What are the trade-offs and results of using the pattern?

GoF Description (cont’d)

- Implementation : pitfalls, hints
- Sample Code : a sketch
- Known Uses
  - Examples of the pattern found in real systems
- Related Patterns
  - What design patterns are closely related to this one?
  - What are the important differences?

Design Pattern Classification

		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory method	Adapter (class)	Interpreter Template method
	Object	Abstract factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Façade Flyweight Proxy	Chain of responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Purpose of a Design Pattern

- Creational patterns:
  - Deal with initializing and configuring classes and objects
- Structural patterns:
  - Deal with decoupling interface and implementation of classes and objects
  - Composition of classes or objects
- Behavioral patterns:
  - Deal with dynamic interactions among societies of classes and objects
  - How they distribute responsibility

Scope of a Design Pattern

- Scope is the domain over which a pattern applies
  - Class Scope: relationships between base classes and their subclasses (static semantics)
  - Object Scope: relationships between peer objects

- Some patterns apply to both scopes.

Architectural Patterns Classification

Category	Architectural Patterns
From Mud to Structure	Layers Pipes and Filters Blackboard
Distributed Systems	Broker Pipes and Filters Microkernel
Interactive Systems	Model-View-Controller (MVC) Presentation-Abstraction-Control (PAC)
Adaptable Systems	Microkernel Reflection

Architectural Patterns Classification (cont’d)

- From Mud to Structure: Support a controlled decomposition of an overall system task into cooperating subtasks
- Distributed Systems: Provides an infrastructure for distributed application
- Interactive Systems: Support the structuring of software systems that feature human-computer interaction
- Adaptable Systems: Support extension of applications and their adaptation to evolving technology and changing functional requirements