Anti Pattern

Identifying bad practices can be as valuable as identifying good practices.

Intro.

- The world of software revolves around maintenance of applications.
- In this complex context, both patterns and antipatterns provide a common vocabulary to software managers, architects, designers, and developers alike.
- Depending on the development team's skill, the product might have either a *good* or a *bad* design.
- Antipattern is the opposite of a design pattern that suggests good design.

Common Antipatterns

Area	Common Antipatterns
Design	Programming to concrete classes rather than interfaces Coupling logic like logging, security and caching in code
Development	Golden Hammer Input Kludge
Architecture	Reinvent the wheel Vendor lock-in

Programming to Concrete Classes Rather than Interfaces

- There will be a provision for changing behavior at runtime.
- Note that this principle is not applicable in situations when you know that the behavior is not going to change.
  An example of programming to an implementation would be:

```
Dog animal = new Dog();

animal.bark();
```

In contrast, programming to an interface would be:

```
Animal animal = getAnimal(Dog.class);

animal.makeNoise(); ……..
```

```
getAnimal(Class c)
 { if (c == Dog.class)
return new Dog(); // other type tests here
 }
class Dog {
makeNoise()
{ bark(); }
}
```

Excessive Coupling

- less coupling is generally better.

For example, if you code a method to perform a certain task, it should perform *only* that task, helping to keep the code clean, readable and maintainable. But inevitably, certain aspects such as logging, security, and caching tend to creep in. There are, fortunately, good ways to avoid this. One such technique, Aspect Oriented Programming (AOP), provides a neat way to achieve this by injecting the aspect's behavior into the application at compile time.

Development: The Golden Hammer

- Excessive or obsessive use of a technology or pattern leads to the Golden Hammer antipattern.
- Teams or individuals well versed in a particular technology or software tend to use that knowledge for any other project of similar nature—even when another technology would be more appropriate. They tend to see unfamiliar technology as risky. In addition, it's easier to plan and estimate with a familiar technology.
- Expanding the knowledge of developers through books, training, and user groups (such as Java User Groups) can be beneficial in avoiding this antipattern.

Input Kludge

- Software that mishandles simple user inputs is an Input Kludge. A Web site that asks users to logon via an ID and password, for example, should accept as input only characters that are valid for the ID and password. If the site logic rejects invalid input, it is fail-safe in that respect, but if it doesn't, unpredictable results can occur. An Input Kludge is an antipattern that can be easily found by the end user, but not so easily during unit testing by the developer.
- use a monkey test to detect an input kludge problem.

Architecture: *Reinventing the Wheel*

- When developing software, you usually have two choices—build on top of an existing technology or start from scratch.
- While both can be applicable in different situations, it is nevertheless useful to analyze existing technology before reinventing the wheel to develop functionality that already exists and can satisfy the requirements. This saves time, money as well as leveraging knowledge that developers already might have.

Vendor lock-in

- This occurs when the software is either partly or wholly dependent on a specific vendor.
- One advantage of J2EE is its portability, but it still gives vendors the opportunity to provide rich proprietary features. Assuredly, such features can aid during development, but they can also have a reverse impact at times. One such problem is loss of control.