Overview of Patterns: Part 1

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt

Professor of Computer Science
Institute for Software Integrated Systems
Vanderbilt University
Nashville, Tennessee, USA
Topics Covered in this Part of the Module

- Motivate the importance of design experience & leveraging recurring design structure in becoming a master software developer
Becoming a Master

- Experts perform differently than beginners
- Unlike novices, professional athletes, musicians & dancers move fluidly & effortlessly, without focusing on each individual movement
Becoming a Master

• Experts perform differently than beginners

• Unlike novices, professional athletes, musicians & dancers move fluidly & effortlessly, without focusing on each individual movement

• When watching experts perform it’s easy to forget how much effort they’ve put into reaching high levels of achievement
Becoming a Master

• Experts perform differently than beginners
  • Unlike novices, professional athletes, musicians & dancers move fluidly & effortlessly, without focusing on each individual movement

• When watching experts perform it’s easy to forget how much effort they’ve put into reaching high levels of achievement

• Continuous repetition & practice are crucial to success
Becoming a Master

- Experts perform differently than beginners
- Unlike novices, professional athletes, musicians & dancers move fluidly & effortlessly, without focusing on each individual movement
- When watching experts perform it’s easy to forget how much effort they’ve put into reaching high levels of achievement
- Continuous repetition & practice are crucial to success
- Mentoring from other experts is also essential to becoming a master
Becoming a Master Software Developer

- Knowledge of programming languages is necessary, but not sufficient
- Can fall prey to “featuritis” or worse
  - e.g., GPERF perfect hash function generator, circa 1990

GPERF
A Perfect Hash Function Generator

Douglas C. Schmidt
schmidt@cs.wustl.edu
http://www.cs.wustl.edu/~schmidt/
Department of Computer Science
Washington University, St. Louis 63130

1 Introduction
Perfect hash functions are a time and space efficient implementation of static search sets. A static search set is an abstract data type (ADT) with operations initialize, insert, and retrieve. Static search sets are common in system software applications. Typical static search sets include compiler and interpreter reserved words, assembler instruction mnemonics, shell interpreter built-in commands, and CORBA IDL compilers. Search set elements are called keywords. Keywords are inserted into the set once, usually off-line at compile-time.

gperf is a freely available perfect hash function generator written in C++ that automatically constructs perfect hash functions from a user-supplied list of keywords. It was designed in the spirit of utilities like lex [1] and yacc [2] to remove the drudgery associated with constructing time and space efficient hash functions.

2 Static Search Sets
There are numerous implementations of static search sets. Common examples include sorted and unsorted arrays and linked lists, AVL trees, optimal binary search trees, digital search trees, deterministic finite-state automata, and various hash table schemes, such as open addressing and bucket chain-
Becoming a Master Software Developer

- Knowledge of programming languages is necessary, but not sufficient
  - Can fall prey to “featuritis” or worse
    - e.g., GPERF perfect hash function generator, circa 1990

See [www.dre.vanderbilt.edu/~schmidt/PDF/gperf.pdf](http://www.dre.vanderbilt.edu/~schmidt/PDF/gperf.pdf) for a paper on GPERF

Problems
- Hard-coded algorithms
- Hard-coded data structures
- Hard-coded generators
  - etc.
Becoming a Master Software Developer

- Knowledge of programming languages is necessary, but not sufficient
- Can fall prey to “featuritis” or worse
  - e.g., GPERF perfect hash function generator, circa 1990

GPERF
A Perfect Hash Function Generator

Douglas C. Schmidt
schmidt@cs.wustl.edu
http://www.cs.wustl.edu/~schmidt/
Department of Computer Science
Washington University, St. Louis 63130

1 Introduction

Perfect hash functions are a time and space efficient implementation of static search sets. A static search set is an abstract data type (ADT) with operations initialize, insert, and retrieve. Static search sets are common in system software applications. Typical static search sets include compiler and interpreter reserved words, assembler instruction mnemonics, shell interpreter built-in commands, and CORBA IDL compilers. Search set elements are called keywords. Keywords are inserted into the set once, usually off-line at compile-time.

GPERF is a freely available perfect hash function generator written in C++ that automatically constructs perfect hash functions from a user-supplied list of keywords. It was designed in the spirit of utilities like lex [1] and yacc [2] to remove the drudgery associated with constructing time and space efficient a sample input keyfile; Section 4 highlights design patterns and implementation strategies used to develop gperf; Section 5 shows the results from empirical benchmarks between gperf-generated recognizers and other popular techniques for reserved word lookup; Section 6 outlines the limitations with gperf and potential enhancements; and Section 7 presents concluding remarks.

2 Static Search Set Implementations

There are numerous implementations of static search sets. Common examples include sorted and unsorted arrays and linked lists, AVL trees, optimal binary search trees, digital search tries, deterministic finite-state automata, and various hash table schemes, such as open addressing and bucket chain-

GPERF is part of the GNU software release at www.gnu.org/software/gperf
Becoming a Master Software Developer

- Knowledge of programming languages is necessary, but not sufficient
- Can fall prey to “featuritis” or worse!
  - e.g., “Best one-liner” from 2006 “Obfuscated C Code” contest

```c
main(_){^448&&main(-~_);putchar(--_%64?32|~7[ TIME _ ~/8%8]["txiZ^(~z?"-48]>>";;=====~$ : 199" [_*2&8]_/64]/(_&2?1:8)%8&1:10);}
```
- This program prints out the time when it was compiled!

See [www.ioccc.org](http://www.ioccc.org) for many examples of obfuscated C
Becoming a Master Software Developer

- Knowledge of programming languages is necessary, but not sufficient
- Can fall prey to “featuritis” or worse!
- Software methods emphasize design notations, such as UML
  - Fine for specification & documentation
    - e.g., omits mundane implementation details & focuses on relationships between key design entities
Becoming a Master Software Developer

- Knowledge of programming languages is necessary, but not sufficient
  - Can fall prey to “featuritis” or worse!
- Software methods emphasize design notations, such as UML
  - Fine for specification & documentation
- But good software design is more than drawing diagrams
  - Good draftsmen/artists are not necessarily good architects!
Becoming a Master Software Developer

- Knowledge of programming languages is necessary, but not sufficient
  - Can fall prey to “featuritis” or worse!
- Software methods emphasize design notations, such as UML
  - Fine for specification & documentation
- But good software design is more than drawing diagrams
  - Good draftsmen/artists are not necessarily good architects!
- **Bottom-line:** Master software developers rely on *design experience*
  - At least as important as knowledge of programming languages & environments

See [www.dre.vanderbilt.edu/~schmidt/PDF/ECOOP-95.pdf](http://www.dre.vanderbilt.edu/~schmidt/PDF/ECOOP-95.pdf) for more info
Where Should Design Experience Reside?

Well-designed software exhibits recurring structures & behaviors that promote:

- Abstraction
- Flexibility
- Reuse
- Quality
- Modularity
Where Should Design Experience Reside?

Well-designed software exhibits recurring structures & behaviors that promote

- Abstraction
- Flexibility
- Reuse
- Quality
- Modularity

Therein lies valuable design knowledge
Where Should Design Experience Reside?

Well-designed software exhibits recurring structures & behaviors that promote:

• Abstraction
• Flexibility
• Reuse
• Quality
• Modularity

Therein lies valuable design knowledge

Unfortunately, this design knowledge is typically located in:

1. the heads of the experts
Where Should Design Experience Reside?

Well-designed software exhibits recurring structures & behaviors that promote:

- Abstraction
- Flexibility
- Reuse
- Quality
- Modularity

Therein lies valuable design knowledge

Unfortunately, this design knowledge is typically located in:

1. the heads of the experts
2. the bowels of the source code
Where Should Design Experience Reside?

Well-designed software exhibits recurring structures & behaviors that promote:

- Abstraction
- Flexibility
- Reuse
- Quality
- Modularity

Therein lies valuable design knowledge.

Unfortunately, this design knowledge is typically located in:

1. the heads of the experts
2. the bowels of the source code

Both locations are fraught with danger!
Summary

• Achieving mastery of software development requires continuous repetition, practice, & mentoring from experts
Summary

- Achieving mastery of software development requires continuous repetition, practice, & mentoring from experts
- Open-source & open courses are vital resources
Summary

- Achieving mastery of software development requires continuous repetition, practice, & mentoring from experts
- Open-source & open courses are vital resources

Information & registration available at [www.coursera.org/course/posa](http://www.coursera.org/course/posa)
Summary

- Achieving mastery of software development requires continuous repetition, practice, & mentoring from experts.

- Good software developers rely on experience gleaned from successful designs.
Summary

- Achieving mastery of software development requires continuous repetition, practice, & mentoring from experts

- Good software developers rely on experience gleaned from successful designs

- What we need is a means of extracting, documenting, conveying, applying, & preserving this design knowledge without undue time, effort, & risk!
Overview of Patterns: Part 2

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt

Professor of Computer Science
Institute for Software Integrated Systems
Vanderbilt University
Nashville, Tennessee, USA
Topics Covered in this Part of the Module

- Motivate the importance of design experience & leveraging recurring design structure in becoming a master software developer
- Introduce patterns as a means of improving software quality & developer productivity by…
Topics Covered in this Part of the Module

- Motivate the importance of design experience & leveraging recurring design structure in becoming a master software developer
- Introduce patterns as a means of improving software quality & developer productivity by…
Topics Covered in this Part of the Module

• Motivate the importance of design experience & leveraging recurring design structure in becoming a master software developer

• Introduce patterns as a means of improving software quality & developer productivity by…
Topics Covered in this Part of the Module

- Motivate the importance of design experience & leveraging recurring design structure in becoming a master software developer
- Introduce patterns as a means of improving software quality & developer productivity by…
Topics Covered in this Part of the Module

- Motivate the importance of design experience & leveraging recurring design structure in becoming a master software developer.
- Introduce patterns as a means of improving software quality & developer productivity by…
Topics Covered in this Part of the Module

- Motivate the importance of design experience & leveraging recurring design structure in becoming a master software developer.

- Introduce patterns as a means of improving software quality & developer productivity.

- Summarize common characteristics of patterns.

```
Subject
state observerList
setData getData notify attach detach

Observer
* update

ConcreteObserver
update doSomething

for all observers in observerList do observer.update()

state = X;
notify();

doSomething
```
Overview of Patterns

Key to Mastery: Knowledge of Software Patterns

• Describes a **solution** to a common **problem** arising within a **context**
Key to Mastery: *Knowledge of Software Patterns*

- Describes a **solution** to a common **problem** arising within a **context** by
- **Naming** a recurring design structure

**Intent:** “Define a one-to-many dependency between objects so that when one object changes state, all dependents are notified & updated”
Key to Mastery: Knowledge of Software Patterns

- Describes a solution to a common problem arising within a context by
  - Naming a recurring design structure
  - Specifying design structure explicitly by identifying key class/object*
    - Roles & relationships
    - Dependencies
    - Interactions
    - Conventions

*Interpret “class” & “object” loosely: patterns are for more than OO languages!
Key to Mastery: *Knowledge of Software Patterns*

- Describes a **solution** to a common **problem** arising within a **context** by
  - **Naming** a recurring design structure
  - **Specifying** design structure explicitly by identifying key class/object
    - Roles & relationships
    - Dependencies
    - Interactions
    - Conventions
  - **Abstracting** from concrete design elements
    - e.g., problem domain, form factor, vendor, etc.

**Observer pattern**

- **Subject**
  - state
  - observerList
  - setData
  - getData
  - notify
  - attach
  - detach

- **Observer**
  - *update

- **ConcreteObserver**
  - update
  - doSomething

- For all observers in observerList do observer.update()
Key to Mastery: *Knowledge of Software Patterns*

- Describes a **solution** to a common **problem** arising within a **context** by
  - **Naming** a recurring design structure
  - **Specifying** design structure explicitly by identifying key class/object
    - Roles & relationships
    - Dependencies
    - Interactions
    - Conventions
  - **Abstracting** from concrete design elements
  - **Distilling & codifying knowledge** gleaned by experts from their successful design experiences
Common Characteristics of Patterns

• They describe both a *thing* & a *process*.
• The “thing” (the “what”) typically means a particular high-level design outline or description of code detail.
Common Characteristics of Patterns

• They describe both a \textit{thing} \& a \textit{process}.

• The “thing” (the “what”) typically means a particular high-level design outline or description of code detail.

• The “process” (the “how”) typically describes the steps to perform to create the “thing”.

\url{csis.pace.edu/~bergin/dcs/SoftwarePatterns_Coplien.pdf} has more info
Common Characteristics of Patterns

- They describe both a *thing* & a *process*
- They can be independent of programming languages & implementation techniques

Naturally, different patterns apply to different programming languages
Common Characteristics of Patterns

- They describe both a *thing* & a *process*
- They can be independent of programming languages & implementation techniques
- They define “micro-architectures”
  - In other words, recurring design structure

```
Subject
  state
  observerList
  attach
detact
  notify
  for all observers in observerList do
  observer.update()

Observer
  update

ConcreteObserver
  update
  ...
```

*Observer pattern*
Common Characteristics of Patterns

• They describe both a *thing* & a *process*
• They can be independent of programming languages & implementation techniques
• They define “micro-architectures”
  • In other words, recurring design structure
• Certain properties may be modified for particular contexts

One use of the *Observer pattern* in Android:

```
Observer

Content Observable

state
observerList

registerObserver
unregisterObserver
notifyChange

for all observers in observerList do
  observer.onChange()

Content Observer

onChange

MyContent Observer

onChange

Concrete Observer

Observer
```
**Common Characteristics of Patterns**

- They describe both a *thing* & a *process*
- They can be independent of programming languages & implementation techniques
- They define “micro-architectures”
  - In other words, recurring design structure
- Certain properties may be modified for particular contexts

A different use of the *Observer pattern* in Android
Common Characteristics of Patterns

- They describe both a *thing* & a *process*
- They can be independent of programming languages & implementation techniques
- They define “micro-architectures”
- They aren’t code or (concrete) designs, so they must be reified and applied in particular languages

**Observer pattern in Java**

```java
public class EventHandler
    extends Observer {
    public void update(Observable o,
        Object arg)
    {
        /*...*/
    }
    ...

    public class EventSource
        extends Observable,
            implements Runnable {
        public void run()
        {
            /*...*/ notifyObservers(/*...*/); }
    ...

    EventSource eventSource =
        new EventSource();
    EventHandler eventHandler =
        new EventHandler();
    eventSource.addObserver(eventHandler);
    Thread thread
        = new Thread(eventSource);
    thread.start();
    ...
```
Overview of Patterns

Douglas C. Schmidt

Common Characteristics of Patterns

- They describe both a *thing* & a *process*
- They can be independent of programming languages & implementation techniques
- They define “micro-architectures”
- They aren’t code or (concrete) designs, so they must be reified and applied in particular languages

```cpp
class Event_Handler
    : public Observer
    
public:
    virtual void update(Observable o, Object arg)
    
    { /* … */ }
    ...

class Event_Source
    : public Observable,
        public ACE_Task_Base
    
public:
    virtual void svc()
    
    { /*…*/ notify_observers(/*…*/); } }
    ...

Event_Source event_source;
Event_Handler event_handler;
event_source->add_observer
    (event_handler);

Event_Task task (event_source);
task->activate();
...
```

*Observer pattern in C++/ACE*

(uses the GoF Bridge pattern with reference counting to simplify memory management & ensure exception-safe semantics)

[www.dre.vanderbilt.edu/ACE has more info on ACE](www.dre.vanderbilt.edu/ACE)
Common Characteristics of Patterns

• They describe both a \textit{thing} \& a \textit{process}

• They can be independent of programming languages \& implementation techniques

• They define “micro-architectures”

• They aren’t code or (concrete) designs, so they must be reified and applied in particular languages

• They are not methods but can be used as an adjunct to methods, e.g.:
  • Rational Unified Process
  • Agile
  • Others
Common Characteristics of Patterns

- They describe both a *thing* & a *process*
- They can be independent of programming languages & implementation techniques
- They define “micro-architectures”
- They aren’t code or (concrete) designs, so they must be reified and applied in particular languages
- They are not methods but can be used as an adjunct to methods
- There are also patterns for organizing effective software development teams and navigating other complex settings
Overview of Patterns

Common Parts of a Pattern Description

• **Name**
  • Should be pithy & memorable

• **Intent**
  • Goal behind the pattern & the reason(s) for using it

• **Problem** addressed by pattern
  • Motivate the “forces” & situations in which pattern is applicable

• **Solution**
  • Visual & textual descriptions of pattern static structure, participants, and collaboration dynamics
Common Parts of a Pattern Description

- **Examples & Implementation guidance**
  - May include source code snippets in one or more programming languages

- **Consequences**
  - Pros & cons of applying the pattern

- **Known uses**
  - Examples of real uses of the pattern
  - Should follow the “rule of three”

- **Related patterns**
  - Summarize relationships & tradeoffs between alternative patterns for similar problems

See [c2.com/cgi/wiki?PatternForms](http://c2.com/cgi/wiki?PatternForms) for more info on pattern forms
Summary

- Patterns codify software expertise & support design at a more abstract level than code
  - Emphasize design *qua* design, not (obscure) language features
  - e.g., the *Observer* pattern can be implemented in many programming languages

Patterns often equated with OO languages, but can apply to non-OO languages
Summary

- Patterns codify software expertise & support design at a more abstract level than code
  - Emphasize design *qua* design, not (obscure) language features
- Treat class/object interactions as a cohesive conceptual unit
  - e.g., form the building blocks for more powerful pattern relationships
Summary

• Patterns codify software expertise & support design at a more abstract level than code
  • Emphasize design *qua* design, not (obscure) language features
  • Treat class/object interactions as a cohesive conceptual unit
• Provide ideal targets for design and implementation refactoring
  • e.g., adapters & (wrapper) facades
Summary

- Stand-alone “pattern islands” are unusual in practice
Summary

- Stand-alone “pattern islands” are unusual in practice
- Patterns are often related & are typically used together
Summary

- Stand-alone “pattern islands” are unusual in practice
- Patterns are often related & are typically used together
- There are various types of pattern relationships
  - Pattern complements

Diagram:

```
+----------------+   +----------------+
| Factory Method |   | Disposal Method |
+----------------+   +----------------+
```
Summary

• Stand-alone “pattern islands” are unusual in practice
• Patterns are often related & are typically used together
• There are various types of pattern relationships
  • Pattern complements
  • Pattern compounds
Overview of Patterns

Summary

- Stand-alone “pattern islands” are unusual in practice
- Patterns are often related & are typically used together
- There are various types of pattern relationships
  - Pattern complements
  - Pattern compounds
  - Pattern sequences
• Stand-alone “pattern islands” are unusual in practice
• Patterns are often related & are typically used together
• There are various types of pattern relationships
  • Pattern complements
  • Pattern compounds
  • Pattern sequences
• Pattern languages

en.wikipedia.org/wiki/Pattern_language has discussions of pattern languages
Summary

- Patterns can be applied in all software lifecycle phases
  - Analysis, design, & reviews
  - Implementation & optimization
  - Testing & documentation
  - Reuse & refactoring
Overview of Patterns: Part 3

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt

Professor of Computer Science
Institute for Software Integrated Systems
Vanderbilt University
Nashville, Tennessee, USA
Topics Covered in this Part of the Module

- Motivate the importance of design experience & leveraging recurring design structure in becoming a master software developer
- Introduce patterns as a means of improving software quality & developer productivity
- Summarize common characteristics of patterns
- Describe a variation-oriented process for successfully applying patterns to software development projects
Variation-oriented Process for Applying Patterns

- To apply patterns successfully, software developers need to:
  - Have broad knowledge of patterns relevant to their domain(s)

See [www.dre.vanderbilt.edu/~schmidt/PDF/ORB-patterns.pdf](http://www.dre.vanderbilt.edu/~schmidt/PDF/ORB-patterns.pdf) for more info
Variation-oriented Process for Applying Patterns

• To apply patterns successfully, software developers need to:

  • Have broad knowledge of patterns relevant to their domain(s)

See c2.com/cgi/wiki?HistoryOfPatterns for a history of patterns
Variation-oriented Process for Applying Patterns

• To apply patterns successfully, software developers need to:
  • Have broad knowledge of patterns relevant to their domain(s)
Variation-oriented Process for Applying Patterns

- To apply patterns successfully, software developers need to:
  - Have broad knowledge of patterns relevant to their domain(s)
  - Evaluate trade-offs & impact of using certain patterns in their software
Variation-oriented Process for Applying Patterns

- To apply patterns successfully, software developers need to:
  - Have broad knowledge of patterns relevant to their domain(s)
  - Evaluate trade-offs & impact of using certain patterns in their software

Mentoring from pattern experts is invaluable, especially when you first start.
Variation-oriented Process for Applying Patterns

- To apply patterns successfully, software developers need to:
  - Have broad knowledge of patterns relevant to their domain(s)
  - Evaluate trade-offs & impact of using certain patterns in their software

Problems
- Hard-coded algorithms
- Hard-coded data structures
- Hard-coded generators
- etc.
Variation-oriented Process for Applying Patterns

• To apply patterns successfully, software developers need to:
  • Have broad knowledge of patterns relevant to their domain(s)
  • Evaluate trade-offs & impact of using certain patterns in their software

Pattern languages help developers navigate thru trade-offs
Variation-oriented Process for Applying Patterns

- To apply patterns successfully, software developers need to:
  - Have broad knowledge of patterns relevant to their domain(s)
  - Evaluate trade-offs & impact of using certain patterns in their software
  - Make design & implementation decisions about how best to apply the selected patterns
  - Patterns may require modifications for particular contexts

The Observer Pattern

```plaintext
Subject
state
observerList
attach
detact
notify

for all observers in observerList do observer.update()

Observer
update

ConcreteObserver
update
...
```
Variation-oriented Process for Applying Patterns

- To apply patterns successfully, software developers need to:
  - Have broad knowledge of patterns relevant to their domain(s)
  - Evaluate trade-offs & impact of using certain patterns in their software
  - Make design & implementation decisions about how best to apply the selected patterns
  - Patterns may require modifications for particular contexts

One use of the Observer Pattern in Android

```java
Observer

Subject

Content Observable

state
observerList

registerObserver
unregisterObserver
notifyChangeEvent

for all observers in observerList do
observer.onChangeEvent()

Content Observer

Observer

MyContent Observer

onChangeEvent
...

Concrete Observer

Observer
```
Variation-oriented Process for Applying Patterns

• To apply patterns successfully, software developers need to:
  • Have broad knowledge of patterns relevant to their domain(s)
  • Evaluate trade-offs & impact of using certain patterns in their software
  • Make design & implementation decisions about how best to apply the selected patterns
  • Patterns may require modifications for particular contexts

A different use of the Observer Pattern in Android

Observer

Subject

Context

state observerList

registerReceiver unregisterReceiver sendBroadcast

for all observers in observerList do observer.onReceive()

Observer

Broadcast Receiver

onReceive

Observer

BroadcastHandler

onReceive

...
Variation-oriented Process for Applying Patterns

- To apply patterns successfully, software developers need to:
  - Have broad knowledge of patterns relevant to their domain(s)
  - Evaluate trade-offs & impact of using certain patterns in their software
  - Make design & implementation decisions about how best to apply the selected patterns
  - Patterns may require modifications for particular contexts

Singleton pattern

```
If (uniqueInstance == 0)
    uniqueInstance = new Singleton;
return uniqueInstance;
```

John Vlissides, “To Kill a Singleton”
[sourcemaking.com/design_patterns/to_kill_a_singleton](sourcemaking.com/design_patterns/to_kill_a_singleton)
Variation-oriented Process for Applying Patterns

- To apply patterns successfully, software developers need to:
  - Have broad knowledge of patterns relevant to their domain(s)
  - Evaluate trade-offs & impact of using certain patterns in their software
  - Make design & implementation decisions about how best to apply the selected patterns
  - Patterns may require modifications for particular contexts

```
class Singleton {
    private static Singleton inst = null;
    public static Singleton instance() {
        Singleton result = inst;
        if (result == null) {
            inst = result = new Singleton();
        }
        return result;
    }
    ...

    static instance() throws Exception {
        return (Singleton) new Singleton();
    }
    singletonOperation() {
        ...
    }
    getSingletonData() {
        ...
    }
    static uniqueInstance {
        ...
    }
    singletonData {
        ...
    }
}
```

Singleton pattern vs. Double-Checked Locking Pattern
Variation-oriented Process for Applying Patterns

- To apply patterns successfully, software developers need to:
  - Have broad knowledge of patterns relevant to their domain(s)
  - Evaluate trade-offs & impact of using certain patterns in their software
  - Make design & implementation decisions about how best to apply the selected patterns
  - Patterns may require modifications for particular contexts

```java
class Singleton {
    private static Singleton inst = null;
    public static Singleton instance() {
        Singleton result = inst;
        if (result == null) {
            inst = result = new Singleton();
        }
        return result;
    }
    ...
}
```

Singleton pattern vs. Double-Checked Locking Pattern

Too little synchronization
Variation-oriented Process for Applying Patterns

- To apply patterns successfully, software developers need to:
  - Have broad knowledge of patterns relevant to their domain(s)
  - Evaluate trade-offs & impact of using certain patterns in their software
  - Make design & implementation decisions about how best to apply the selected patterns
  - Patterns may require modifications for particular contexts

<table>
<thead>
<tr>
<th>Singleton</th>
</tr>
</thead>
<tbody>
<tr>
<td>static instance()</td>
</tr>
<tr>
<td>singletonOperation()</td>
</tr>
<tr>
<td>getSingletonData()</td>
</tr>
<tr>
<td>static uniqueInstance</td>
</tr>
<tr>
<td>singletonData</td>
</tr>
</tbody>
</table>

If (uniqueInstance == 0)
  uniqueInstance = new Singleton;
  return uniqueInstance;

Singleton pattern vs. Double-Checked Locking Pattern

class Singleton {
    private static Singleton inst = null;
    public static Singleton instance() {
        synchronized(Singleton.class) {
            Singleton result = inst;
            if (result == null) {
                inst = result = new Singleton();
            }
        }
        return result;
    }
    ...
}
Variation-oriented Process for Applying Patterns

• To apply patterns successfully, software developers need to:
  • Have broad knowledge of patterns relevant to their domain(s)
  • Evaluate trade-offs & impact of using certain patterns in their software
  • Make design & implementation decisions about how best to apply the selected patterns
  • Patterns may require modifications for particular contexts

If (uniqueInstance == 0)
uniqueInstance = new Singleton;
return uniqueInstance;

Singleton pattern vs. Double-Checked Locking Pattern

class Singleton {
    private static Singleton inst = null;
    public static Singleton instance() {
        synchronized(Singleton.class) {
            Singleton result = inst;
            if (result == null) {
                inst = result = new Singleton();
            }
        }
        return result;
    }
}

Too much synchronization
Variation-oriented Process for Applying Patterns

- To apply patterns successfully, software developers need to:
  - Have broad knowledge of patterns relevant to their domain(s)
  - Evaluate trade-offs & impact of using certain patterns in their software
  - Make design & implementation decisions about how best to apply the selected patterns
  - Patterns may require modifications for particular contexts

```
class Singleton {
    private static volatile Singleton inst = null;
    public static Singleton instance() {
        Singleton result = inst;
        if (result == null) {
            synchronized(Singleton.class) {
                result = inst;
                if (result == null) {
                    inst = result = new Singleton();
                }
            }
        }
        return result;
    }
}
```

Just right amount of synchronization

Singleton pattern vs. Double-Checked Locking Pattern

```java
If (uniqueInstance == 0)
    uniqueInstance = new Singleton;
return uniqueInstance;
```
Variation-oriented Process for Applying Patterns

- To apply patterns successfully, software developers need to:
  - Have broad knowledge of patterns relevant to their domain(s)
  - Evaluate trade-offs & impact of using certain patterns in their software
  - Make design & implementation decisions about how best to apply the selected patterns
  - Patterns may require modifications for particular contexts

```java
class Singleton {
    private static volatile Singleton inst = null;

    public static Singleton instance() {
        Singleton result = inst;
        if (result == null) {
            synchronized(Singleton.class) {
                result = inst;
                if (result == null) {
                    inst = result = new Singleton();
                }
            }
        }
        return result;
    }
    ...
}
```

**Singleton pattern vs. Double-Checked Locking Pattern**

Only synchronizes when inst is null
Variation-oriented Process for Applying Patterns

To apply patterns successfully, software developers need to:

- Have broad knowledge of patterns relevant to their domain(s)
- Evaluate trade-offs & impact of using certain patterns in their software
- Make design & implementation decisions about how best to apply the selected patterns
- Patterns may require modifications for particular contexts

```java
class Singleton {
    private static volatile Singleton inst = null;

    public static Singleton instance() {
        Singleton result = inst;
        if (result == null) {
            synchronized(Singleton.class) {
                result = inst;
                if (result == null) {
                    inst = result = new Singleton();
                }
            }
        }
        return result;
    }
}
```
Overview of Patterns

Variation-oriented Process for Applying Patterns

- To apply patterns successfully, software developers need to:
  - Have broad knowledge of patterns relevant to their domain(s)
  - Evaluate trade-offs & impact of using certain patterns in their software
  - Make design & implementation decisions about how best to apply the selected patterns
  - Patterns may require modifications for particular contexts

<table>
<thead>
<tr>
<th>Singleton</th>
</tr>
</thead>
<tbody>
<tr>
<td>static instance()</td>
</tr>
<tr>
<td>singletonOperation()</td>
</tr>
<tr>
<td>getSingletonData()</td>
</tr>
<tr>
<td>static uniqueInstance</td>
</tr>
<tr>
<td>singletonData</td>
</tr>
</tbody>
</table>

If (uniqueInstance == 0)
uniqueInstance = new Singleton;
return uniqueInstance;

Singleton pattern vs. Double-Checked Locking Pattern

class Singleton {
    private static volatile Singleton inst = null;
    public static Singleton instance() {
        Singleton result = inst;
        if (result == null) {
            synchronized(Singleton.class) {
                result = inst;
                if (result == null) {
                    inst = result = new Singleton();
                }
            }
        }
        return result;
    }
    ...
}

See en.wikipedia.org/wiki/Double-checked_locking for more info
Overview of Patterns

Variation-oriented Process for Applying Patterns

• To apply patterns successfully, software developers need to:
  • Have broad knowledge of patterns relevant to their domain(s)
  • Evaluate trade-offs & impact of using certain patterns in their software
  • Make design & implementation decisions about how best to apply the selected patterns
    • Patterns may require modifications for particular contexts
  • Combine with other patterns & implement/integrate with code
Variation-oriented Process for Applying Patterns

- To apply patterns successfully, software developers need to:
  - Have broad knowledge of patterns relevant to their domain(s)
  - Evaluate trade-offs & impact of using certain patterns in their software
  - Make design & implementation decisions about how best to apply the selected patterns
  - Patterns may require modifications for particular contexts
  - Combine with other patterns & implement/integrate with code
Patterns support a variation-oriented design process

1. Determine which design elements can vary
2. Identify applicable pattern(s)
3. Vary patterns & evaluate trade-offs
4. Repeat…
Overview of Patterns

Summary

• Seek generality, but don’t brand everything as a pattern
Summary

• Seek generality, but don’t brand everything as a pattern

• Articulate specific benefits and demonstrate general applicability
  • e.g., find three different existing examples from code other than yours!