

# Software Architecture (2/2)

Martin Kellogg

# Software Architecture (Part 2 of 3 2)

Today's agenda:

- Reading Quiz
- Strategies for good design
- Design patterns
  - Structural patterns
  - Creational patterns
  - Behavioural patterns

# Software Architecture (Part 1)

Today's agenda:

- Reading Quiz
- Strategies for good design
- Design patterns
  - Structural patterns
  - Creational patterns
  - Behavioural patterns

Announcements:

- I will accept optional reading responses via email until end of day tomorrow for up to  $\frac{1}{2}$  credit
- Sprint 2 is over; you should have your sprint 2 mentor meeting before EoD Wednesday

# Software Architecture (Part 2 of 3 2)

Today's agenda:

- **Reading Quiz**
- Strategies for good design
- Design patterns
  - Structural patterns
  - Creational patterns
  - Behavioural patterns

# Reading quiz: architecture 2

Q1: Which of the following does the author claim are important parts of design patterns?

- A. Patterns must have an “evocative” name
- B. Patterns must be solutions to recurring problems
- C. Both A and B
- D. Neither A nor B

Q2: **TRUE** or **FALSE**: the author argues that an advantage of a microservice architecture is that it organizes teams around projects or technologies rather than products

# Reading quiz: architecture 2

Q1: Which of the following does the author claim are important parts of design patterns?

- A. Patterns must have an “evocative” name
- B. Patterns must be solutions to recurring problems
- C. Both A and B
- D. Neither A nor B

Q2: **TRUE** or **FALSE**: the author argues that an advantage of a microservice architecture is that it organizes teams around projects or technologies rather than products

# Reading quiz: architecture 2

Q1: Which of the following does the author claim are important parts of design patterns?

- A. Patterns must have an “evocative” name
- B. Patterns must be solutions to recurring problems
- C. Both A and B
- D. Neither A nor B

Q2: **TRUE** or **FALSE**: the author argues that an advantage of a microservice architecture is that it organizes teams around projects or technologies rather than products

# Software Architecture (Part 2 of 3 2)

Today's agenda:

- Reading Quiz
- **Strategies for good design**
- Design patterns
  - Structural patterns
  - Creational patterns
  - Behavioural patterns



Design

# “Architecture” vs “Design”

Development process



Requirements

Architecture

Design

Source Code



Level of Abstraction

**Definition:** *software design* is the structure or organization of a particular component of your system

# Design

Key goal: design for **change** and **reuse**

# Design

**Key goal:** design for **change** and **reuse**

- In class, many programs are written once, to a fixed specification, and then **thrown away**

# Design

**Key goal:** design for **change** and **reuse**

- In class, many programs are written once, to a fixed specification, and then **thrown away**
- In industry, many programs are written once and then **modified** as requirements, customers, and developers change

# Design

**Key goal:** design for **change** and **reuse**

- In class, many programs are written once, to a fixed specification, and then **thrown away**
- In industry, many programs are written once and then **modified** as requirements, customers, and developers change
- Many fundamental tenets of object-oriented design facilitate **subsequent change**
  - You may have seen these before, but now you are in a position to really appreciate the motivation!

Design: desiderata

# Design: desiderata

- Classes are **open** to be modified/extended without invasive changes



# Design: desiderata

- Classes are **open** to be modified/extended without invasive changes
- **Subtype polymorphism** enables changes behind interfaces

# Design: desiderata

- Classes are **open** to be modified/extended without invasive changes
- **Subtype polymorphism** enables changes behind interfaces
- Classes **encapsulate** details likely to change behind (small) stable interfaces

# Design: desiderata

- Classes are **open** to be modified/extended without invasive changes
- **Subtype polymorphism** enables changes behind interfaces
- Classes **encapsulate** details likely to change behind (small) stable interfaces
- Internal parts can be developed **independently**

# Design: desiderata

- Classes are **open** to be modified/extended without invasive changes
- **Subtype polymorphism** enables changes behind interfaces
- Classes **encapsulate** details likely to change behind (small) stable interfaces
- Internal parts can be developed **independently**
- Internal details of other classes do not need to be understood, **contract** is sufficient

# Design: desiderata

- Classes are **open** to be modified/extended without invasive changes
- **Subtype polymorphism** enables changes behind interfaces
- Classes **encapsulate** details likely to change behind (small) stable interfaces
- Internal parts can be developed **independently**
- Internal details of other classes do not need to be understood, **contract** is sufficient
- Class implementations and their contracts can be tested separately (**unit testing**)

# Design for reuse

# Design for reuse: delegation

**Definition:** *Delegation* is when one object relies on another object for some subset of its functionality

- e.g., in Java, Sort delegates functionality to some Comparator

# Design for reuse: delegation

**Definition:** *Delegation* is when one object relies on another object for some subset of its functionality

- e.g., in Java, Sort delegates functionality to some Comparator

Judicious delegation enables **code reuse**:



# Design for reuse: delegation

**Definition:** *Delegation* is when one object relies on another object for some subset of its functionality

- e.g., in Java, Sort delegates functionality to some Comparator

Judicious delegation enables **code reuse**:

- Sort can be reused with arbitrary sort orders

# Design for reuse: delegation

**Definition:** *Delegation* is when one object relies on another object for some subset of its functionality

- e.g., in Java, Sort delegates functionality to some Comparator

Judicious delegation enables **code reuse**:

- Sort can be reused with arbitrary sort orders
- Comparators can be reused with arbitrary client code that needs to compare integers

# Design for reuse: delegation

**Definition:** *Delegation* is when one object relies on another object for some subset of its functionality

- e.g., in Java, Sort delegates functionality to some Comparator

Judicious delegation enables **code reuse**:

- Sort can be reused with arbitrary sort orders
- Comparators can be reused with arbitrary client code that needs to compare integers
- Reduce “cut and paste” code and defects

# Design for change: motivation

- Amazon.com processes millions of orders each year, selling in 75 countries, all 50 states, and thousands of cities worldwide.
  - These countries, states, and cities have hundreds of distinct sales tax policies
  - For any order and destination, Amazon.com must be able to compute the correct sales tax for the order and destination.

# Design for change: motivation

Our key goal for today:  
learn about some of the  
**strategies** that companies  
like Amazon.com use to  
manage this complexity

- Amazon.com processes millions of orders across many countries, all 50 states, and thousands of cities worldwide.
  - These countries, states, and cities have hundreds of distinct sales tax policies
  - For any order and destination, Amazon.com must be able to compute the correct sales tax for the order and destination.
- Over time:
  - Amazon moves into new markets
  - Laws and taxes in existing markets change

# Design for Extensibility: Contracts + Subtyping

# Design for Extensibility: Contracts + Subtyping

- **Design by contract** prescribes that software designers should define formal, precise and **verifiable** interface specifications for components, which extend the ordinary definition of abstract data types with **preconditions, postconditions** and invariants

# Design for Extensibility: Contracts + Subtyping

- **Design by contract** prescribes that software designers should define formal, precise and **verifiable** interface specifications for components, which extend the ordinary definition of abstract data types with **preconditions, postconditions** and invariants
- A subclass can only have **weaker** preconditions
  - My super only works on positive numbers, but I work on all numbers



# Design for Extensibility: Contracts + Subtyping

- **Design by contract** prescribes that software designers should define formal, precise and **verifiable** interface specifications for components, which extend the ordinary definition of abstract data types with **preconditions, postconditions** and invariants
- A subclass can only have **weaker** preconditions
  - My super only works on positive numbers, but I work on all numbers
- A subclass can only have **stronger** postconditions
  - My super returns any shape, but I return squares

# Design for Extensibility: Contracts + Subtyping

- **Design by contract** prescribes that software designers should define formal, precise and **verifiable** interface specifications for components, which extend the ordinary definition of abstract data types with **preconditions, postconditions**
- A subclass can only have **weaker** preconditions
  - My super only works on positive numbers
- A subclass can only have **stronger** postconditions
  - My super returns any shape, but I return squares

This is called the **Liskov Substitution Principle**: “any subclass object should be safe to use in place of a super class object at run time”

# Design for Testability

# Design for Testability

- If the majority cost of software engineering is maintenance

# Design for Testability

- If the majority cost of software engineering is maintenance
  - and the majority cost of maintenance is QA

# Design for Testability

- If the majority cost of software engineering is maintenance
  - and the majority cost of maintenance is QA
  - and the majority cost of QA is testing

# Design for Testability

- If the majority cost of software engineering is maintenance
  - and the majority cost of maintenance is QA
  - and the majority cost of QA is testing
- Then we should design our software so that **testing** is effective:

# Design for Testability

- If the majority cost of software engineering is maintenance
  - and the majority cost of maintenance is QA
  - and the majority cost of QA is testing
- Then we should design our software so that **testing** is effective:
  - Design to admit testing
  - Design to admit fault injection
  - Design to admit coverage
  - Recognize “free test” opportunities



# Design to Admit Testing

# Design to Admit Testing

- Consider a *library-oriented architecture*, a variation of **modular programming** or **microservice architecture** with a focus on separation of concerns and **interface design**

# Design to Admit Testing

- Consider a *library-oriented architecture*, a variation of **modular programming** or **microservice architecture** with a focus on separation of concerns and **interface design**
  - “Package logical components of your application independently - literally as separate gems, eggs, RPMs, or whatever - and maintain them as internal open-source projects ... This approach combats the tightly-coupled spaghetti so often lurking in big codebases by giving everything the Right Place in which to exist.”

# Design to Admit Unit Testing

- Recall: it is hard to generate test inputs with high coverage for areas “**deep inside**” the code
  - Must solve the constraints for main(), then for foo(), then for bar(), etc., all at the same time!

# Design to Admit Unit Testing

- Recall: it is hard to generate test inputs with high coverage for areas “**deep inside**” the code
  - Must solve the constraints for main(), then for foo(), then for bar(), etc., all at the same time!
- The farther code is from an entry point, the harder it is to test
  - This is one of the motivations behind **unit testing**

# Design to Admit Unit Testing

- Recall: it is hard to generate test inputs with high coverage for areas “**deep inside**” the code
  - Must solve the constraints for main(), then for foo(), then for bar(), etc., all at the same time!
- The farther code is from an entry point, the harder it is to test
  - This is one of the motivations behind **unit testing**
- Solution: design with **more entry points** for self-contained functionality (cf. AVL tree, priority queue, etc.)

# Example: MVC + Angry Birds



- Suppose you are designing Angry Birds
- It's a game, and also a simulation, so MVC is a reasonable choice

# Example: MVC + Angry Birds



- Suppose you are designing Angry Birds
- It's a game, and also a simulation, so MVC is a reasonable choice
- Design so that it can be tested without someone actually playing the game!



# Example: MVC + Angry Birds



- Suppose you are designing Angry Birds
- It's a game, and also a simulation, so MVC is a reasonable choice
- Design so that it can be tested without someone actually playing the game!
  - e.g., have an **interface** where abstract commands can be queued up: one way to get them is from the UI, but another is programmatic

# Example: MVC + Angry Birds



- Suppose you are designing Angry Birds
- It's a game, and also a simulation, so MVC is a reasonable choice
- Design so that it can be tested without someone actually playing the game!
  - e.g., have an **interface** where abstract commands can be queued up: one way to get them is from the UI, but another is programmatic
  - “If I create a world with blocks X, Y and Z and then we launch bird A at angle B, does C occur within five timesteps?”

# Example: fault injection

- Microsoft's Driver Verifier sat between a driver and the operating system and “pretended to fail (some of the time)” to expose poor driver code
- The CHES project sat between a program and the scheduler and “forced strange schedules” to expose poor concurrency code

# Example: fault injection

- Microsoft's Driver Verifier sat between a driver and the operating system and “pretended to fail (some of the time)” to expose poor driver code
- The CHES project sat between a program and the scheduler and “forced strange schedules” to expose poor concurrency code
- Problem for both: Hardware, OS and Networking errors can occur **infrequently**, but you still want to test them

# Example: fault injection

- Microsoft's Driver Verifier sat between a driver and the operating system and “pretended to fail (some of the time)” to expose poor driver code
- The CHES project sat between a program and the scheduler and “forced strange schedules” to expose poor concurrency code
- Problem for both: Hardware, OS and Networking errors can occur **infrequently**, but you still want to test them
  - Must design for it! But how...?

# Example: fault injection: add a level of indirection

- **Old adage**: the solution to everything in computer science is either to add a level of indirection or to add a cache

# Example: fault injection: add a level of indirection

- **Old adage**: the solution to everything in computer science is either to add a level of indirection or to add a cache
- Don't have your code call `fopen()` or `cout` or whatever directly

# Example: fault injection: add a level of indirection

- **Old adage**: the solution to everything in computer science is either to add a level of indirection or to add a cache
- Don't have your code call `fopen()` or `cout` or whatever directly
- Instead, add a very thin **level of indirection** where you call `my_fopen` which then calls `fopen`



# Example: fault injection: add a level of indirection

- **Old adage**: the solution to everything in computer science is either to add a level of indirection or to add a cache
- Don't have your code call `fopen()` or `cout` or whatever directly
- Instead, add a very thin **level of indirection** where you call `my_fopen` which then calls `fopen`
- Later add “if `coin_flip()` then fail else ...” to that indirection layer to **inject faults** while testing
  - let the compiler optimize it away for your production code

# Designing for coverage-based testing

- Remind me: what's **coverage** (in the context of testing)?

# Designing for coverage-based testing

- Remind me: what's **coverage** (in the context of testing)?
- Code coverage has many flaws
  - At a high level, simple coverage metrics **do not align** with covering requirements (cf. traceability)

# Designing for coverage-based testing

- Remind me: what's **coverage** (in the context of testing)?
- Code coverage has many flaws
  - At a high level, simple coverage metrics **do not align** with covering requirements (cf. traceability)
- Solutions:

# Designing for coverage-based testing

- Remind me: what's **coverage** (in the context of testing)?
- Code coverage has many flaws
  - At a high level, simple coverage metrics **do not align** with covering requirements (cf. traceability)
- Solutions:
  - Better test suite adequacy metrics (mutation, etc.)

# Designing for coverage-based testing

- Remind me: what's **coverage** (in the context of testing)?
- Code coverage has many flaws
  - At a high level, simple coverage metrics **do not align** with covering requirements (cf. traceability)
- Solutions:
  - Better test suite adequacy metrics (mutation, etc.)
  - Design and write the code so that high code coverage **correlates** with high requirements coverage!

# Designing for coverage-based testing

- Line coverage is often inadequate because “visit line 5 when ptr==null” could be very different from “visit line 5 when ptr !=null”

# Designing for coverage-based testing

- Line coverage is often inadequate because “visit line 5 when ptr==null” could be very different from “visit line 5 when ptr !=null”
  - Because “\*ptr = 9” is really “if (ptr == null) abort(); else \*ptr = 9;”



# Designing for coverage-based testing

- Line coverage is often inadequate because “visit line 5 when ptr==null” could be very different from “visit line 5 when ptr !=null”
  - Because “\*ptr = 9” is really “if (ptr == null) abort(); else \*ptr = 9;”
- Consider **explicit conditionals** that check **requirements adherence**

# Designing for coverage-based testing

- Line coverage is often inadequate because “visit line 5 when ptr==null” could be very different from “visit line 5 when ptr !=null”
  - Because “\*ptr = 9” is really “if (ptr == null) abort(); else \*ptr = 9;”
- Consider **explicit conditionals** that check **requirements adherence**
  - To get coverage points for reaching the true branch, the test will have to satisfy the requirement

# Designing for coverage-based testing

- Line coverage is often inadequate because “visit line 5 when ptr==null” could be very different from “visit line 5 when ptr !=null”
  - Because “\*ptr = 9” is really “if (ptr == null) abort(); else \*ptr = 9;”
- Consider **explicit conditionals** that check **requirements adherence**
  - To get coverage points for reaching the true branch, the test will have to satisfy the requirement
- For example, consider a quality requirement: “finish X within Y time”

# Designing for coverage-based testing

- Line coverage is often inadequate because “visit line 5 when `ptr==null`” could be very different from “visit line 5 when `ptr !=null`”
  - Because “`*ptr = 9`” is really “if (`ptr == null`) abort(); else `*ptr = 9;`”
- Consider **explicit conditionals** that check **requirements adherence**
  - To get coverage points for reaching the true branch, the test will have to satisfy the requirement
- For example, consider a quality requirement: “finish X within Y time”
  - Add in “get the time”, “do X”, “get the time”, “subtract”, “if `t2 - t1 < Y` then ...”

# Designing for coverage-based testing

- Line coverage is often inadequate
  - `ptr==null` could be very difficult to test
    - Because `*ptr = 9` is real
- Consider **explicit conditional**
  - To get coverage points for `ptr==null` you have to satisfy the requirement
- For example, consider a quality requirement
  - Add in “get the time”, “do X”
  - Y then ...”

Explicit Conditional **Pros:**

Explicit Conditional **Cons:**

# Designing for coverage-based testing

- Line coverage is often inadequate
  - `ptr==null` could be very difficult to test
    - Because `*ptr = 9` is real
- Consider **explicit conditional**
  - To get coverage points for `ptr==null` you have to satisfy the requirement
- For example, consider a quality requirement
  - Add in “get the time”, “do X if Y then ...”

## Explicit Conditional **Pros:**

- Testing tools can help you reason about partial progress

## Explicit Conditional **Cons:**

# Designing for coverage-based testing

- Line coverage is often inadequate
  - `ptr==null` could be very difficult to test
    - Because `*ptr = 9` is real
- Consider **explicit conditional**
  - To get coverage points for `ptr==null` you have to satisfy the requirement
- For example, consider a quality requirement
  - Add in “get the time”, “do X”, “do Y then ...”

## Explicit Conditional **Pros:**

- Testing tools can help you reason about partial progress
- Testing tools can try to falsify claims

## Explicit Conditional **Cons:**

# Designing for coverage-based testing

- Line coverage is often inadequate
  - `ptr==null` could be very difficult to test
    - Because `*ptr = 9` is real
- Consider **explicit conditional**
  - To get coverage points for `ptr==null` you have to satisfy the requirement
- For example, consider a quality requirement
  - Add in “get the time”, “do X”, “do Y then ...”

## Explicit Conditional **Pros**:

- Testing tools can help you reason about partial progress
- Testing tools can try to falsify claims

## Explicit Conditional **Cons**:

- Muddies meaning of coverage (100% not desired)



Designing for testing: tests for free

# Designing for testing: tests for free

- Many programs transform data from one format to another (cf. adapter pattern)

# Designing for testing: tests for free

- Many programs transform data from one format to another (cf. adapter pattern)
- If the program is implementing a function with similar domain and range, you can often get high-coverage tests “for free” by **composing the program with itself**

# Designing for testing: tests for free

- Many programs transform data from one format to another (cf. adapter pattern)
- If the program is implementing a function with similar domain and range, you can often get high-coverage tests “for free” by **composing the program with itself**
  - If possible, design your program so that this is possible

# Designing for testing: tests for free

- Many programs transform data from one format to another (cf. adapter pattern)
- If the program is implementing a function with similar domain and range, you can often get high-coverage tests “for free” by **composing the program with itself**
  - If possible, design your program so that this is possible
- Examples:

# Designing for testing: tests for free

- Many programs transform data from one format to another (cf. adapter pattern)
- If the program is implementing a function with similar domain and range, you can often get high-coverage tests “for free” by **composing the program with itself**
  - If possible, design your program so that this is possible
- Examples:
  - **Inversion**: for all  $X$ .  $\text{unzip}(\text{zip}(X)) = X$

# Designing for testing: tests for free

- Many programs transform data from one format to another (cf. adapter pattern)
- If the program is implementing a function with similar domain and range, you can often get high-coverage tests “for free” by **composing the program with itself**
  - If possible, design your program so that this is possible
- Examples:
  - **Inversion**:  $\text{forall } X. \text{unzip}(\text{zip}(X)) = X$
  - **Convergence**:  $\text{forall } X. \text{sort}(\text{sort}(X)) = \text{sort}(X)$

# Software Architecture (Part 2 of 3 2)

Today's agenda:

- Reading Quiz
- Strategies for good design
- **Design patterns**
  - Structural patterns
  - Creational patterns
  - Behavioural patterns



# Design patterns

# Design patterns

**Definition:** A *software design pattern* is a general, reusable solution to a commonly occurring problem within a given context in software design.

# Design patterns

**Definition:** A *software design pattern* is a general, reusable solution to a commonly occurring problem within a given context in software design.

- patterns reduce surprise for code readers

# Design patterns

**Definition:** A *software design pattern* is a general, reusable solution to a commonly occurring problem within a given context in software design.

- patterns reduce surprise for code readers
- patterns separate the structure of a system from the implementation details

# Design patterns

**Definition:** A *software design pattern* is a general, reusable solution to a commonly occurring problem within a given context in software design.

- patterns reduce surprise for code readers
- patterns separate the structure of a system from the implementation details
- patterns apply in almost all OO languages

# Design patterns

**Definition:** A *software design pattern* is a general, reusable solution to a commonly occurring problem within a given context in software design.

- patterns reduce surprise for code readers
- patterns separate the structure of a system from the implementation details
- patterns apply in almost all OO languages
- all patterns have **tradeoffs**. In OO languages, design patterns often trade **verbosity or efficiency** for **extensibility**

# Design patterns

**Definition:** A *software design pattern* is a general, reusable solution to a commonly occurring problem within a given context in software design.

- patterns reduce surprise for code readers
- patterns separate the structure of a system from the implementation details
- patterns apply in almost all OO languages
- all patterns have **tradeoffs**. In OO languages, design patterns often trade **verbosity or efficiency** for **extensibility**
- we'll consider **structural**, **creational** and **behavioral** design patterns

# Design patterns: non-software

- Design patterns are **common in almost every field** (not just SE)



# Design patterns: non-software

- Design patterns are **common in almost every field** (not just SE)
  - e.g., “multiple choice question” is a design pattern for exam making, “verse-chorus-verse-chorus-bridge-chorus” is a design pattern for songs

# Design patterns: non-software

- Design patterns are **common in almost every field** (not just SE)
  - e.g., “multiple choice question” is a design pattern for exam making, “verse-chorus-verse-chorus-bridge-chorus” is a design pattern for songs
- *The Design of Everyday Things* (famous book): design serves as the **communication** between object and user

# Design patterns: non-software

- Design patterns are **common in almost every field** (not just SE)
  - e.g., “multiple choice question” is a design pattern for exam making, “verse-chorus-verse-chorus-bridge-chorus” is a design pattern for songs
- *The Design of Everyday Things* (famous book): design serves as the **communication** between object and user
  - although people often blame themselves when objects appear to malfunction, it is not the fault of the user but rather the lack of intuitive guidance that should be present in the design

# Design patterns: non-software

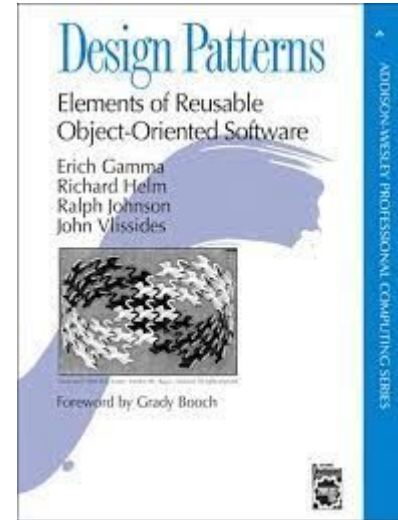
- Design patterns are **common**
  - e.g., “multiple choice question making,” “verse-chorus-verse” pattern for songs
- *The Design of Everyday Things* (Norman, 1988) is the **communication** between object and user
  - although people often blame themselves when objects appear to malfunction, it is not the fault of the user but rather the lack of intuitive guidance that should be present in the design

Same ideas apply to software:

- design GUIs that people **intuitively** know how to use
- design code that other developers **intuitively** know how to read

# Design patterns: “gang of four”

- The book popularizing software design patterns is often called the “**Gang of Four**” book after its four authors
- I don’t care if you remember this, but it’ll be handy to know about (e.g., for interviews)



Design patterns: high-level advice

# Design patterns: high-level advice

- Consider **code change as a certainty**
  - Redesign is expensive. Choosing the right pattern helps avoid it.

# Design patterns: high-level advice

- Consider **code change as a certainty**
  - Redesign is expensive. Choosing the right pattern helps avoid it.
- Consider your **requirements and their changes**
  - Use patterns that fit your current or anticipated needs.



# Design patterns: high-level advice

- Consider **code change as a certainty**
  - Redesign is expensive. Choosing the right pattern helps avoid it.
- Consider your **requirements and their changes**
  - Use patterns that fit your current or anticipated needs.
- Consider **multiple designs**
  - Diagram your designs before writing code.

# Design patterns: categories

- We're going to talk about three **categories** of design patterns:
  - Structural
  - Creational
  - Behavioural

# Design patterns: categories

- We're going to talk about three **categories** of design patterns:
  - Structural
  - Creational
  - Behavioural
- These are **rough** categories: some patterns don't fit neatly into any of them, and some might fit into two or more
  - but, they're a useful way to structure a lecture :)

# Design patterns: categories

- We're going to talk about three **categories** of design patterns:
  - **Structural**
  - Creational
  - Behavioural
- These are **rough** categories: some patterns don't fit neatly into any of them, and some might fit into two or more
  - but, they're a useful way to structure a lecture :)

# Design patterns: structural

- *Structural design patterns* ease design by identifying simple ways to realize **relationships** among **entities**.

# Design patterns: structural

- *Structural design patterns* ease design by identifying simple ways to realize **relationships** among **entities**.
- In software, they usually:

# Design patterns: structural

- *Structural design patterns* ease design by identifying simple ways to realize **relationships** among **entities**.
- In software, they usually:
  - Build new classes or interfaces from existing ones

# Design patterns: structural

- *Structural design patterns* ease design by identifying simple ways to realize **relationships** among **entities**.
- In software, they usually:
  - Build new classes or interfaces from existing ones
  - Hide implementation details



# Design patterns: structural

- *Structural design patterns* ease design by identifying simple ways to realize **relationships** among **entities**.
- In software, they usually:
  - Build new classes or interfaces from existing ones
  - Hide implementation details
  - Provide cleaner or more specialized interfaces

# Design patterns: structural: adapter

- The *adapter design pattern* is a structural design pattern that converts the interface of a class into another interface clients expect.

# Design patterns: structural: adapter

- The *adapter design pattern* is a structural design pattern that converts the interface of a class into another interface clients expect.
  - analogy: dongles that convert HDMI to USB-C

# Design patterns: structural: adapter

- The *adapter design pattern* is a structural design pattern that converts the interface of a class into another interface clients expect.
  - analogy: dongles that convert HDMI to USB-C
- Examples:

# Design patterns: structural: adapter

- The *adapter design pattern* is a structural design pattern that converts the interface of a class into another interface clients expect.
  - analogy: dongles that convert HDMI to USB-C
- Examples:
  - Implementing a Stack interface using a LinkedList interface

# Design patterns: structural: adapter

- The *adapter design pattern* is a structural design pattern that converts the interface of a class into another interface clients expect.
  - analogy: dongles that convert HDMI to USB-C
- Examples:
  - Implementing a Stack interface using a LinkedList interface
  - Early implementations of fstream in C++
    - ... were simply adapters around the C FILE macro

# Design patterns: other structural patterns

# Design patterns: other structural patterns

- The **composite design pattern** allows clients to treat individual objects and groups of objects uniformly



# Design patterns: other structural patterns

- The **composite design pattern** allows clients to treat individual objects and groups of objects uniformly
  - e.g., selecting and moving objects in PowerPoint

# Design patterns: other structural patterns

- The **composite design pattern** allows clients to treat individual objects and groups of objects uniformly
  - e.g., selecting and moving objects in PowerPoint
- The **proxy design pattern** provides a surrogate or placeholder for another object to control access to it

# Design patterns: other structural patterns

- The **composite design pattern** allows clients to treat individual objects and groups of objects uniformly
  - e.g., selecting and moving objects in PowerPoint
- The **proxy design pattern** provides a surrogate or placeholder for another object to control access to it
  - e.g., `std::vector` exposes `std::vector::reference` as a method of accessing individual bits. In particular, objects of this class are returned by `operator[]` by value.

[https://en.cppreference.com/w/cpp/container/vector\\_bool](https://en.cppreference.com/w/cpp/container/vector_bool)

# Design patterns: creational patterns

# Design patterns: creational patterns

- *Creational design patterns* avoid complexity by controlling object creation so that objects are created in a manner suitable for the situation. They make a system **independent of how its objects are created**.

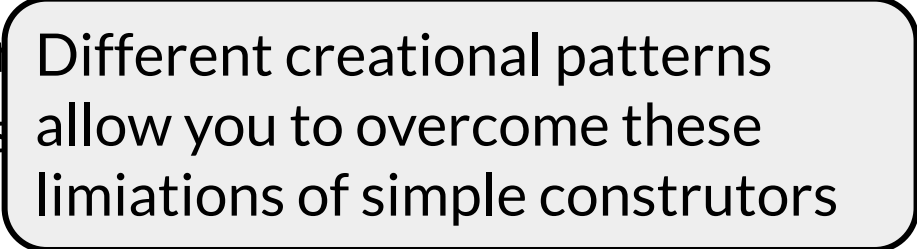
# Design patterns: creational patterns

- *Creational design patterns* avoid complexity by controlling object creation so that objects are created in a manner suitable for the situation. They make a system **independent of how its objects are created**.
- A plain constructor **may not allow** you to:

# Design patterns: creational patterns

- *Creational design patterns* avoid complexity by controlling object creation so that objects are created in a manner suitable for the situation. They make a system **independent of how its objects are created**.
- A plain constructor **may not allow** you to:
  - Control how and when an object is used
  - Overcome language limitations (e.g., no default arguments)
  - Hide polymorphic types
  - Specify different combinations of optional arguments

# Design patterns: creational patterns

- **Creational design patterns** avoid complexity by controlling object creation so that objects are created in a consistent manner. They make a system **are created**.  

- A plain constructor **may not allow** you to:
  - Control how and when an object is used
  - Overcome language limitations (e.g., no default arguments)
  - Hide polymorphic types
  - Specify different combinations of optional arguments



# Creational patterns: named constructor

- In the *Named Constructor Pattern*, you declare the class's normal constructors to be private or protected and make a public static creation method.

# Creational patterns: named constructor

- In the *Named Constructor Pattern*, you declare the class's normal constructors to be private or protected and make a public static creation method.

```
class Llama {  
public:  
    static Llama* create_llama(string name) {  
        return new Llama(name);  
    }  
private: // Making ctor private  
    Llama(string name_in): name(name_in) {}  
    string name;  
};
```

# Creational patterns: named constructor

- In the *Named Constructor Pattern*, constructors to be private or protected and a named creation method.

```
class Llama {  
public:  
    static Llama* create_llama(string name)  
    {  
        return new Llama(name);  
    }  
private: // Making ctor private  
    Llama(string name_in): name(name_in) {}  
    string name;  
};
```

Why might you do this?

- might want to change to Llama subclass later
- want to validate arguments from clients, but make construction fast internally
- etc.

# Creational patterns: factories

- Suppose we need to create and use polymorphic objects **without exposing their types** to the client

# Creational patterns: factories

- Suppose we need to create and use polymorphic objects **without exposing their types** to the client
  - Recall: design for maintainability and extensibility. We don't want the client to depend on (and thus “lock in”) the actual subtypes.

# Creational patterns: factories

- Suppose we need to create and use polymorphic objects **without exposing their types** to the client
  - Recall: design for maintainability and extensibility. We don't want the client to depend on (and thus “lock in”) the actual subtypes.
- The typical solution is to write a function that creates objects of the type we want but returns that object so that it appears to be (“cast to”) a member of the base class

# Creational patterns: factories

- Suppose we need to create and use polymorphic objects **without exposing their types** to the client
  - Recall: design for maintainability and extensibility. We don't want the client to depend on (and thus “lock in”) the actual subtypes.
- The typical solution is to write a function that creates objects of the type we want but returns that object so that it appears to be (“cast to”) a member of the base class
  - this is a specific variant of the named constructor pattern

# Creational patterns: factories

- The *factory method pattern* (or just *factory pattern*) is a creational design pattern that uses factory methods to create objects without having the return type reveal the exact subclass created.



# Creational patterns: factories

- The *factory method pattern* (or just *factory pattern*) is a creational design pattern that uses factory methods to create objects without having the return type reveal the exact subclass created.

```
Payment * payment_factory(string name, string type) {  
    if (type == "credit_card")  
        return new CreditCardPayment(name);  
    else if (type == "bitcoin")  
        return new BitcoinPayment(name);  
    ... }  

```

```
Payment * webapp_session_payment =  
    payment_factory(customer_name, "credit_card");  

```

# Creational patterns: factories

- The *factory method pattern* (or design pattern that uses factories without having the return type

Note how the implementation details are hidden from the client, and they can only treat the result as a **generic** payment

```
Payment * payment_factory(string name, string type) {  
    if (type == "credit_card")  
        return new CreditCardPayment(name);  
    else if (type == "bitcoin")  
        return new BitcoinPayment(name);  
    ... }  

```

```
Payment * webapp_session_payment =  
    payment_factory(customer_name, "credit_card");  

```

# Creational patterns: factories

- You may also encounter implementations in which special methods create the right type:

# Creational patterns: factories


- You may also encounter implementations in which special methods create the right type:


```
class PaymentFactory {
public:
    static Payment* make_credit_payment(string name){
        return new CreditCardPayment(name);
    }
    static Payment* make_bc_payment(string name){
        return new BitcoinPayment(name);
    }
};

Payment * webapp_session_payment =
PaymentFactory::make_credit_payment(customer_name);
```

# Creational patterns: example

- Suppose we're implementing a computer game with a **polymorphic Enemy class hierarchy**, and we want to spawn **different versions** of enemies based on the difficulty level.

- e.g., normal difficulty = regular Goomba 

- hard difficulty = spiked Goomba 

# Creational patterns: example: anti-patterns

- An *anti-pattern* is a common response to a recurring problem that is usually ineffective and risks being counterproductive.

# Creational patterns: example: anti-patterns

- An *anti-pattern* is a common response to a recurring problem that is usually ineffective and risks being counterproductive.
- A bad solution (i.e., anti-pattern) would be to check the difficulty at each of the many places in the code related to spawning enemies:

```
Enemy* goomba = nullptr;
if (difficulty == "normal")
    goomba = new Goomba();
else if (difficulty == "hard")
    goomba = new SpikedGoomba();
```

# Creational patterns: example: anti-patterns

- An *anti-pattern* is a common response to a recurring problem that is usually ineffective and risks being counterproductive.
- A bad solution (i.e., anti-pattern) would be to check the difficulty at each of the many places in the code related to spawning enemies:

```
Enemy* goomba = nullptr;
if (difficulty == "normal")
    goomba = new Goomba();
else if (difficulty == "hard")
    goomba = new SpikedGoomba();
```

Why is this bad?



# Creational patterns: example: anti-patterns

- An *anti-pattern* is a common response to a recurring problem that is usually ineffective and risks being counterproductive.
- A bad solution (i.e., anti-pattern) would be to check the difficulty at each of the many places in the code related to spawning enemies:

```
Enemy* goomba = nullptr;
if (difficulty == "normal")
    goomba = new Goomba();
else if (difficulty == "hard")
    goomba = new SpikedGoomba();
```

## Why is this bad?

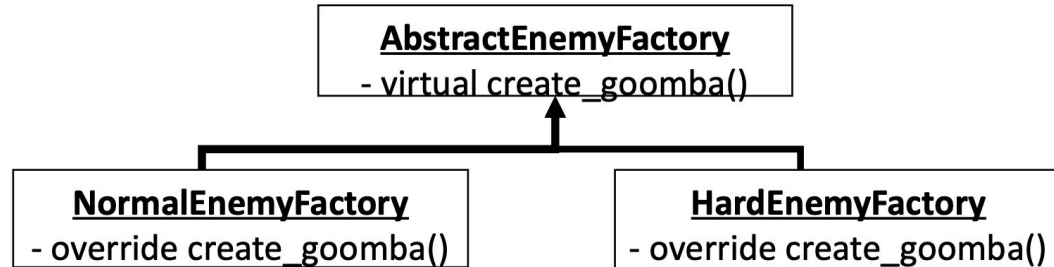
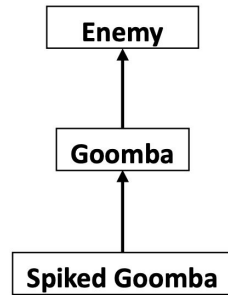
- code duplication
- consider how you'd add a new difficulty level...

# Creational patterns: abstract factories

- The *abstract factory pattern* encapsulates a group of factories that have a common theme without specifying their concrete classes.

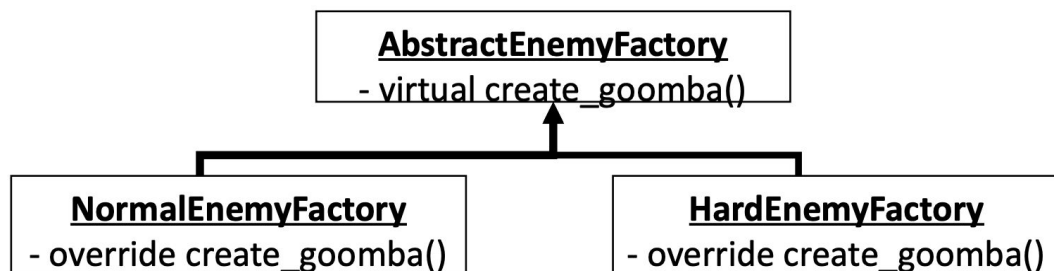
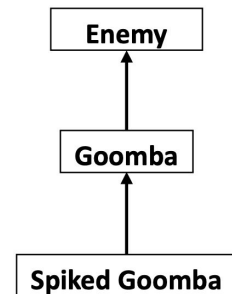
# Creational patterns: abstract factories

- The *abstract factory pattern* encapsulates a group of factories that have a common theme without specifying their concrete classes.



# Creational patterns: abstract factories

- The **abstract factory pattern** encapsulates a group of factories that have a common theme without specifying their concrete classes.



```
// Only have to do this once!  
AbstractEnemyFactory* factory = nullptr;  
if (difficulty == "normal")  
    factory = new NormalEnemyFactory();  
else if (difficulty == "hard")  
    factory = new HardEnemyFactory();  
Enemy* goomba = factory->create_goomba();
```

Scenario: global application state

# Scenario: global application state

- Suppose we have some application **state that needs to be globally accessible**. However, we need to control how that data is accessed and updated.

# Scenario: global application state

- Suppose we have some application **state that needs to be globally accessible**. However, we need to control how that data is accessed and updated.
- The anti-pattern (**bad**) solution is to have an **unprotected global variable** (e.g., a public static field).

# Scenario: global application state

- Suppose we have some application **state that needs to be globally accessible**. However, we need to control how that data is accessed and updated.
- The anti-pattern (**bad**) solution is to have an **unprotected global variable** (e.g., a public static field).
  - fails to control access or updates!



# Scenario: global application state

- Suppose we have some application **state that needs to be globally accessible**. However, we need to control how that data is accessed and updated.
- The anti-pattern (**bad**) solution is to have an **unprotected global variable** (e.g., a public static field).
  - fails to control access or updates!
- A “less bad” solution is to put all of the state in one class and have a **global instance** of that class.

# Scenario: global application state

- Global variables are usually a **poor design choice**. However:

# Scenario: global application state

- Global variables are usually a **poor design choice**. However:
  - If you **must** access some state everywhere, passing it as a parameter to every function clutters the code (readability vs. ...)

# Scenario: global application state

- Global variables are usually a **poor design choice**. However:
  - If you **must** access some state everywhere, passing it as a parameter to every function clutters the code (readability vs. ... )
    - This is not an argument for using global variables to avoid passing a few parameters.

# Scenario: global application state

- Global variables are usually a **poor design choice**. However:
  - If you **must** access some state everywhere, passing it as a parameter to every function clutters the code (readability vs. ... )
    - This is not an argument for using global variables to avoid passing a few parameters.
  - Or if you need to access state stored outside your program (e.g., database, web API)

# Scenario: global application state

- Global variables are usually a **poor design choice**. However:
  - If you **must** access some state everywhere, passing it as a parameter to every function clutters the code (readability vs. ... )
    - This is not an argument for using global variables to avoid passing a few parameters.
  - Or if you need to access state stored outside your program (e.g., database, web API)
  - Then global variables **may** be acceptable

# Singleton design pattern

- The *singleton pattern* restricts the instantiation of a class to **exactly one** logical instance. It ensures that a class has only one logical instance at runtime and provides a global point of access to it.

## Singleton

public:

- static ***get\_instance()*** // *named ctor*

private:

- static ***instance*** // *the one instance*

- Singleton() // *ctor*

# Singleton design pattern: example

```
class Singleton {
    // public way to get "the one logical instance"
    public static Singleton get_instance() {
        if (Singleton.instance == null) Singleton.instance = new Singleton();
        return Singleton.instance;
    }
    private static Singleton instance = null;
    private Singleton() { // only runs once
        billing_database = 0;
        System.out.println("Singleton DB created");
    }
    // Our global state
    private int billing_database;
    public int get_billing_count() { return billing_database; }
    public void increment_billing_count() { billing_database += 1; }
}
```



# Singleton design pattern: example

```
class Singleton {
    // public way to get "the one logical instance"
    public static Singleton get_instance() {
        if (Singleton.instance == null) Singleton.instance = new Singleton();
        return Singleton.instance;
    }
    private static Singleton instance = null;
    private Singleton() { // only runs once
        billing_database = 0;
        System.out.println("Singleton DB created");
    }
    // Our global state
    private int billing_database;
    public int get_billing_count() { return billing_database; }
    public void increment_billing_count() { billing_database += 1; }
}
```

lazy initialization  
of single object



# Singleton design pattern: example

```
class Singleton {
    // public way to get "the one logical instance"
    public static Singleton get_instance() {
        if (Singleton.instance == null) Singleton.instance = new Singleton();
        return Singleton.instance;
    }
    private static Singleton instance = null;
    private Singleton() { // only runs once
        billing_database = 0;
        System.out.println("Singleton DB created");
    }
    // Our global state
    private int billing_database;
    public int get_billing_count() { return billing_database; }
    public void increment_billing_count() { billing_database += 1; }
}
```

this constructor  
can't be called any  
other way



# Singleton design pattern: example

```
class Singleton {
    // public way to get "the one logical instance"
    public static Singleton get_instance() {
        if (Singleton.instance == null) Singleton.instance = new Singleton();
        return Singleton.instance;
    }
    private static Singleton instance = null;
    private Singleton() { // only runs once
        billing_database = 0;
        System.out.println("Singleton DB created");
    }
    // Our global state
    private int billing_database;
    public int get_billing_count() { return billing_database; }
    public void increment_billing_count() { billing_database += 1; }
}
```

all clients share  
this global state



# Singleton design pattern: example

What is the output of this code?

```
class Main {  
    public static void main(String[] args) {  
        int bills = Singleton.get_instance().get_billing_count();  
        System.out.println(bills);  
  
        Singleton.get_instance().increment_billing_count();  
        bills = Singleton.get_instance().get_billing_count();  
        System.out.println(bills);  
    }  
}
```

## Singleton

public:

- static ***get\_instance()*** // *named ctor*
- *get\_billing\_count()*
- *increment\_billing\_count()* // *adds 1*

private:

- static ***instance*** // *the one instance*
- *Singleton()* // *ctor, prints message*
- *billing\_database*

# Singleton design pattern: example

What is the output of this code?

```
class Main {  
    public static void main(String[] args) {  
        int bills = Singleton.get_instance().get_billing_count();  
        System.out.println(bills);  
  
        Singleton.get_instance().increment_billing_count();  
        bills = Singleton.get_instance().get_billing_count();  
        System.out.println(bills);  
    }  
}
```

## Singleton

public:

- static **get\_instance()** // named ctor
- get\_billing\_count()
- increment\_billing\_count() // adds 1

private:

- static **instance** // the one instance
- Singleton() // ctor, prints message
- billing\_database

## Output:

```
Singleton DB created  
0  
1
```

# Singleton design pattern: get\_instance()

- Could we avoid typing `Single.get_instance()` so many times by doing this at all of the points in our program that use the singleton?

```
Single s = Singleton.get_instance();  
System.out.println(s.get_billing_count());  
... // later  
System.out.println(s.get_billing_count());
```

# Singleton design pattern: get\_instance()

- Could we avoid typing `Single.get_instance()` so many times by doing this at all of the points in our program that use the singleton?

```
Single s = Singleton.get_instance();  
System.out.println(s.get_billing_count());  
... // later  
System.out.println(s.get_billing_count());
```

- Is this a good idea or not?

# Singleton design pattern: get\_instance()

- Could we avoid typing `Single.get_instance()` so many times by doing this at all of the points in our program that use the singleton?

```
Single s = Singleton.get_inst  
System.out.println(s.get_bill  
... // later  
System.out.println(s.get_bill
```

- Is this a good idea or not?

This is a **bad idea**. There is **no guarantee** that `get_instance()` will return the same pointer (same object) every time it is called. (It may return different **concrete copies** of the **same logical item**.)



# Singleton design pattern: another example

- Suppose we are implementing a computer version of the card game Euchre. In addition to a few abstract datatypes, we have a Game class that stores the state needed for a game of Euchre. When started, our application prototype plays one game of Euchre and then exits.
- Design question: **should we make Game a singleton?**

# Singleton design pattern: another example

- Making Game a Singleton is **tempting**
  - There is only one Game instance in our application

# Singleton design pattern: another example

- Making Game a Singleton is **tempting**
  - There is only one Game instance in our application
- However, there only **happens** to be one instance of Game. There's **no requirement** that we only have one instance.

# Singleton design pattern: another example

- Making Game a Singleton is **tempting**
  - There is only one Game instance in our application
- However, there only **happens** to be one instance of Game. There's **no requirement** that we only have one instance.
- We should only use the Singleton pattern when current or future **requirements** dictate that only one instance should exist.

# Singleton design pattern: another example

- Making Game a Singleton is **tempting**
  - There is only one Game instance in our application
- However, there only **happens** to be one instance of Game. There's **no requirement** that we only have one instance.
- We should only use the Singleton pattern when current or future **requirements** dictate that only one instance should exist.
  - Singleton is **not** a license to make everything global.

# Behavioural Design Patterns

# Behavioural Design Patterns

- *Behavioral design patterns* support common communication patterns among objects. They are concerned with algorithms and the assignment of responsibilities between objects.

# Behavioural Design Patterns

- *Behavioral design patterns* support common communication patterns among objects. They are concerned with algorithms and the assignment of responsibilities between objects.
  - Commonly used to enable **limited sharing**



# Behavioural Design Patterns

- *Behavioral design patterns* support common communication patterns among objects. They are concerned with algorithms and the assignment of responsibilities between objects.
  - Commonly used to enable **limited sharing**
    - e.g., same underlying algorithm, different interfaces or same interface, different underlying algorithms

# Behavioural Design Patterns

- *Behavioral design patterns* support common communication patterns among objects. They are concerned with algorithms and the assignment of responsibilities between objects.
  - Commonly used to enable **limited sharing**
    - e.g., same underlying algorithm, different interfaces or same interface, different underlying algorithms
  - Examples: strategy pattern, template method pattern, iterator pattern, observer pattern, etc.

# Iterator Pattern

- The *iterator pattern* is a common behavioral design pattern. It provides a uniform interface for traversing containers regardless of how they are implemented.

# Iterator Pattern

- The *iterator pattern* is a common behavioral design pattern. It provides a uniform interface for traversing containers regardless of how they are implemented.
  - e.g., Java's List interface doesn't care whether it's backed by an array or a linked list

# Iterator Pattern

- The *iterator pattern* is a common behavioral design pattern. It provides a uniform interface for traversing containers regardless of how they are implemented.
  - e.g., Java's List interface doesn't care whether it's backed by an array or a linked list
- Similar patterns exist for other kinds of data structures
  - e.g., *visitor pattern* for tree-like structures

# Strategy Design Pattern

# Strategy Design Pattern

- Problem: Clients need different **variants** of an algorithm

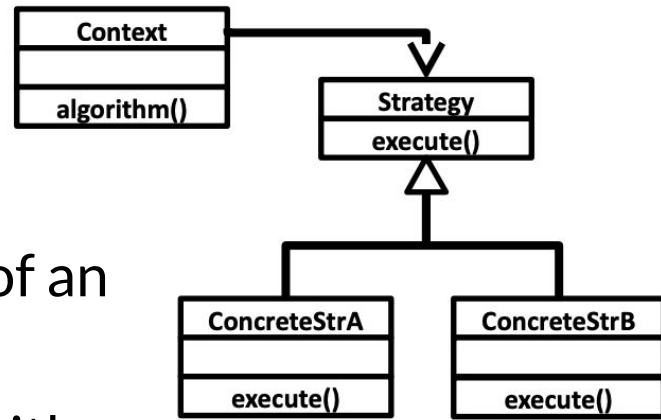
# Strategy Design Pattern

- Problem: Clients need different **variants** of an algorithm
- Solution: Create an **interface** for the algorithm, with an implementing class for each variant of the algorithm



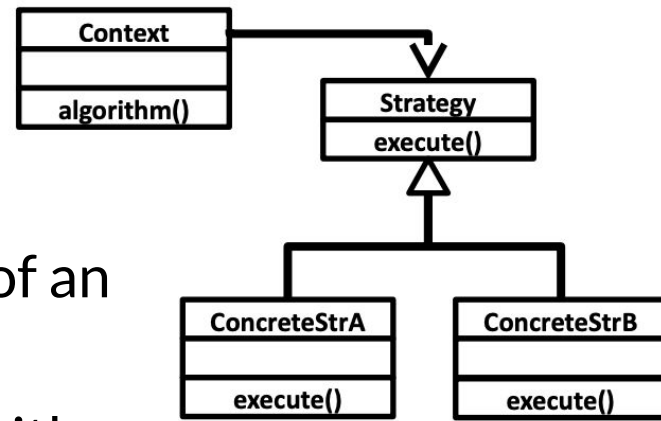
# Strategy Design Pattern

- Problem: Clients need different **variants** of an algorithm
- Solution: Create an **interface** for the algorithm, with an implementing class for each variant of the algorithm



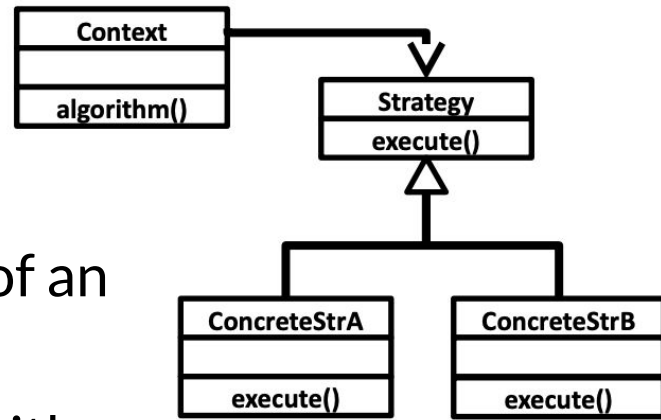
# Strategy Design Pattern

- Problem: Clients need different **variants** of an algorithm
- Solution: Create an **interface** for the algorithm, with an implementing class for each variant of the algorithm
- Consequences:



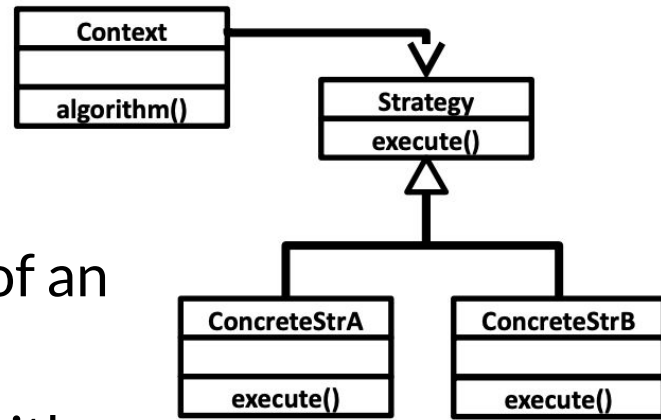
# Strategy Design Pattern

- Problem: Clients need different **variants** of an algorithm
- Solution: Create an **interface** for the algorithm, with an implementing class for each variant of the algorithm
- Consequences:
  - Easily extensible for new algorithm implementations



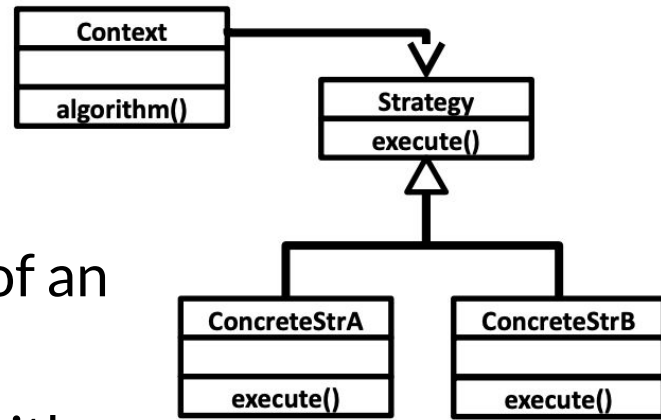
# Strategy Design Pattern

- Problem: Clients need different **variants** of an algorithm
- Solution: Create an **interface** for the algorithm, with an implementing class for each variant of the algorithm
- Consequences:
  - Easily extensible for new algorithm implementations
  - Separates algorithm from client context



# Strategy Design Pattern

- Problem: Clients need different **variants** of an algorithm
- Solution: Create an **interface** for the algorithm, with an implementing class for each variant of the algorithm
- Consequences:
  - Easily extensible for new algorithm implementations
  - Separates algorithm from client context
  - Introduces extra interfaces and classes: code can be harder to understand; adds overhead if the strategies are simple



# Template Method Design Pattern

# Template Method Design Pattern

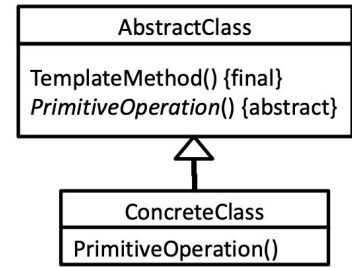
- Problem: An algorithm has **customizable** and **invariant** parts

# Template Method Design Pattern

- Problem: An algorithm has **customizable** and **invariant** parts
- Solution: Implement the invariant parts of the algorithm in an **abstract** class, with abstract primitive operations representing the customizable parts of the algorithm. Subclasses customize the primitive operations.

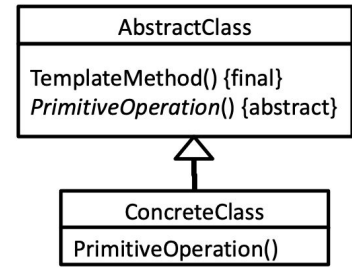


# Template Method Design Pattern



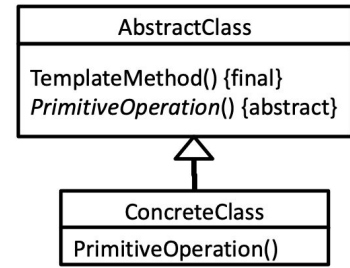
- Problem: An algorithm has **customizable** and **invariant** parts
- Solution: Implement the invariant parts of the algorithm in an **abstract** class, with abstract primitive operations representing the customizable parts of the algorithm. Subclasses customize the primitive operations.

# Template Method Design Pattern



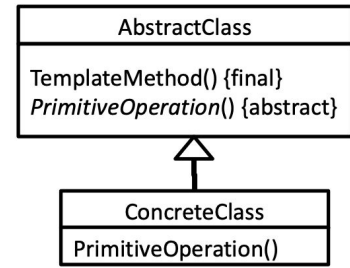
- Problem: An algorithm has **customizable** and **invariant** parts
- Solution: Implement the invariant parts of the algorithm in an **abstract** class, with abstract primitive operations representing the customizable parts of the algorithm. Subclasses customize the primitive operations.
- Consequences:

# Template Method Design Pattern



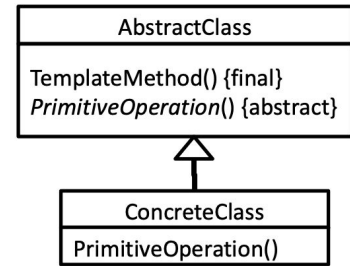
- Problem: An algorithm has **customizable** and **invariant** parts
- Solution: Implement the invariant parts of the algorithm in an **abstract** class, with abstract primitive operations representing the customizable parts of the algorithm. Subclasses customize the primitive operations.
- Consequences:
  - Code reuse for the invariant parts of algorithm

# Template Method Design Pattern



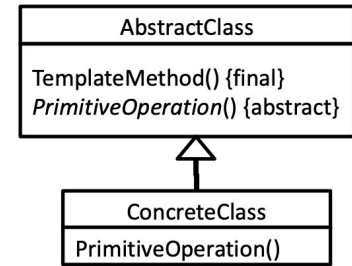
- Problem: An algorithm has **customizable** and **invariant** parts
- Solution: Implement the invariant parts of the algorithm in an **abstract** class, with abstract primitive operations representing the customizable parts of the algorithm. Subclasses customize the primitive operations.
- Consequences:
  - Code reuse for the invariant parts of algorithm
  - Customization is restricted to the primitive operations

# Template Method Design Pattern



- Problem: An algorithm has **customizable** and **invariant** parts
- Solution: Implement the invariant parts of the algorithm in an **abstract** class, with abstract primitive operations representing the customizable parts of the algorithm. Subclasses customize the primitive operations.
- Consequences:
  - Code reuse for the invariant parts of algorithm
  - Customization is restricted to the primitive operations
  - Inverted (“Hollywood-style”) control for customization: “don’t call us, we’ll call you” (cf. comparison function in sorting)

# Template Method Design Pattern



- Problem: An algorithm has **customizable** and **invariant** parts
- Solution: Implement the invariant parts of the algorithm in an **abstract** class, with abstract primitive operations representing the customizable parts of the algorithm. Subclasses customize the primitive operations.
- Consequences:
  - Code reuse for the invariant parts of algorithm
  - Customization is restricted to the primitive operations
  - Inverted (“Hollywood-style”) control for customization: “don’t call us, we’ll call you” (cf. comparison function in sorting)
  - Invariant parts of the algorithm are not changed by subclasses

# Template vs. Strategy Design Pattern

# Template vs. Strategy Design Pattern

- Both support variation in a larger context



# Template vs. Strategy Design Pattern

- Both support variation in a larger context
- **Template method** uses inheritance + an overridable method

# Template vs. Strategy Design Pattern

- Both support variation in a larger context
- **Template method** uses inheritance + an overridable method
- **Strategy** uses an interface and polymorphism (via composition)
  - Strategy objects are reusable across multiple classes
  - Multiple strategy objects are possible per class

# Scenario: binge-watching

- Suppose we're implementing a video streaming website in which users can “binge-watch” (or “lock on”) to one channel. The user will then see that channel's videos in sequence. When the last such video is watched, the user should stop binge-watching that channel.

# Scenario: binge-watching

- Idea: when the last video is watched, call `release_binge_watch()` on the user.

# Scenario: binge-watching

- Idea: when the last video is watched, call `release_binge_watch()` on the user.

```
class User {  
    public void release_binge_watch(Channel c) {  
        if (c == binge_channel) {  
            binge_channel = null;  
        }  
    }  
    private Channel binge_channel;  
}
```

# Scenario: binge-watching

- Idea: when the last video is watched, call `release_binge_watch()` on the user.

```
class User {  
    public void release_binge_watch(Channel c) {  
        if (c == binge_channel) {  
            binge_channel = null;  
        }  
    }  
    private Channel binge_channel;  
}
```

```
class Channel {  
    // Called when the last video is shown  
    public void on_last_video_shown() {  
        // Global accessor for the user  
        get_user().release_binge_watch(this);  
    }  
}
```

# Scenario: binge-watching

- Idea: when the last video is watched, call `release_binge_watch()` on the user.

```
class User {  
    public void release_binge_watch(Channel c) {  
        if (c == binge_channel) {  
            binge_channel = null;  
        }  
    }  
    private Channel binge_channel;  
}
```

```
class Channel {  
    // Called when the last video is shown  
    public void on_last_video_shown() {  
        // Global accessor for the user  
        get_user().release_binge_watch(this);  
    }  
}
```

- What are some problems with this approach?

# Scenario: binge-watching: anti-patterns

- With this design, User and Channel are **tightly coupled**
  - Changing one likely requires a change to the other



# Scenario: binge-watching: anti-patterns

- With this design, User and Channel are **tightly coupled**
  - Changing one likely requires a change to the other
- The design does not support multiple users

# Scenario: binge-watching: anti-patterns

- With this design, User and Channel are **tightly coupled**
  - Changing one likely requires a change to the other
- The design does not support multiple users
- What if we later want to update a user's “recommendation queue” when they finish binge-watching a channel?

# Scenario: binge-watching: anti-patterns

- With this design, User and Channel are **tightly coupled**
  - Changing one likely requires a change to the other
- The design does not support multiple users
- What if we later want to update a user's “recommendation queue” when they finish binge-watching a channel?
- Whenever requirements change and we want to do something else when a video finishes (e.g., update advertising) we **must update the Channel class** and couple it to the new feature

# Scenario: binge-watching: anti-patterns

- With this design, User and Channel are **tightly coupled**
  - Changing one likely requires a change to the other
- The design does not allow for a “recommendation queue”
- What if we later want to add a “recommendation queue” when they finish binge-watching a channel?
- Whenever requirements change and we want to do something else when a video finishes (e.g., update advertising) we **must update the Channel class** and couple it to the new feature

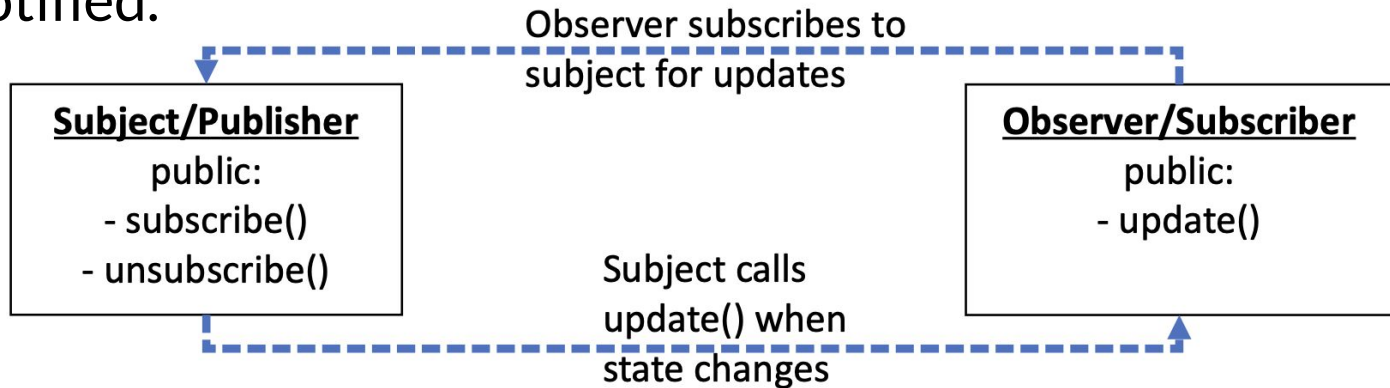
What can we do instead?

# Observer Pattern

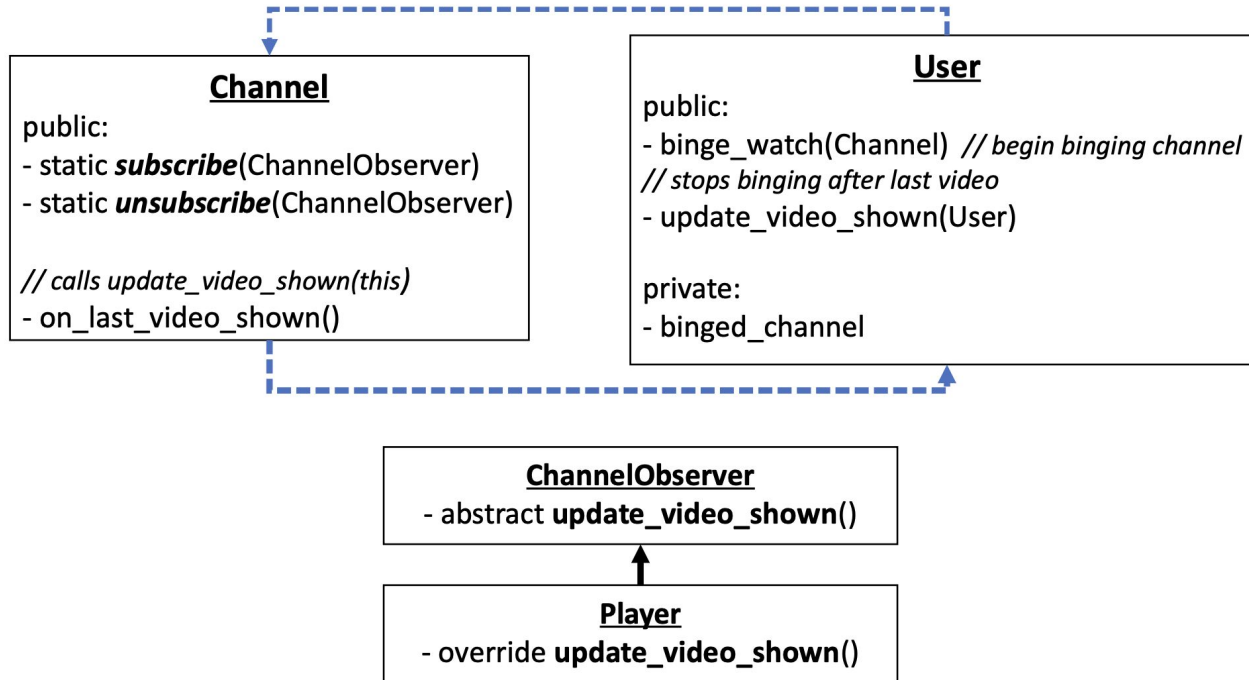
- The *observer pattern* (also called “*publish-subscribe*”) allows dependent objects to be notified automatically when the state of a subject changes. It defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified.

# Observer Pattern

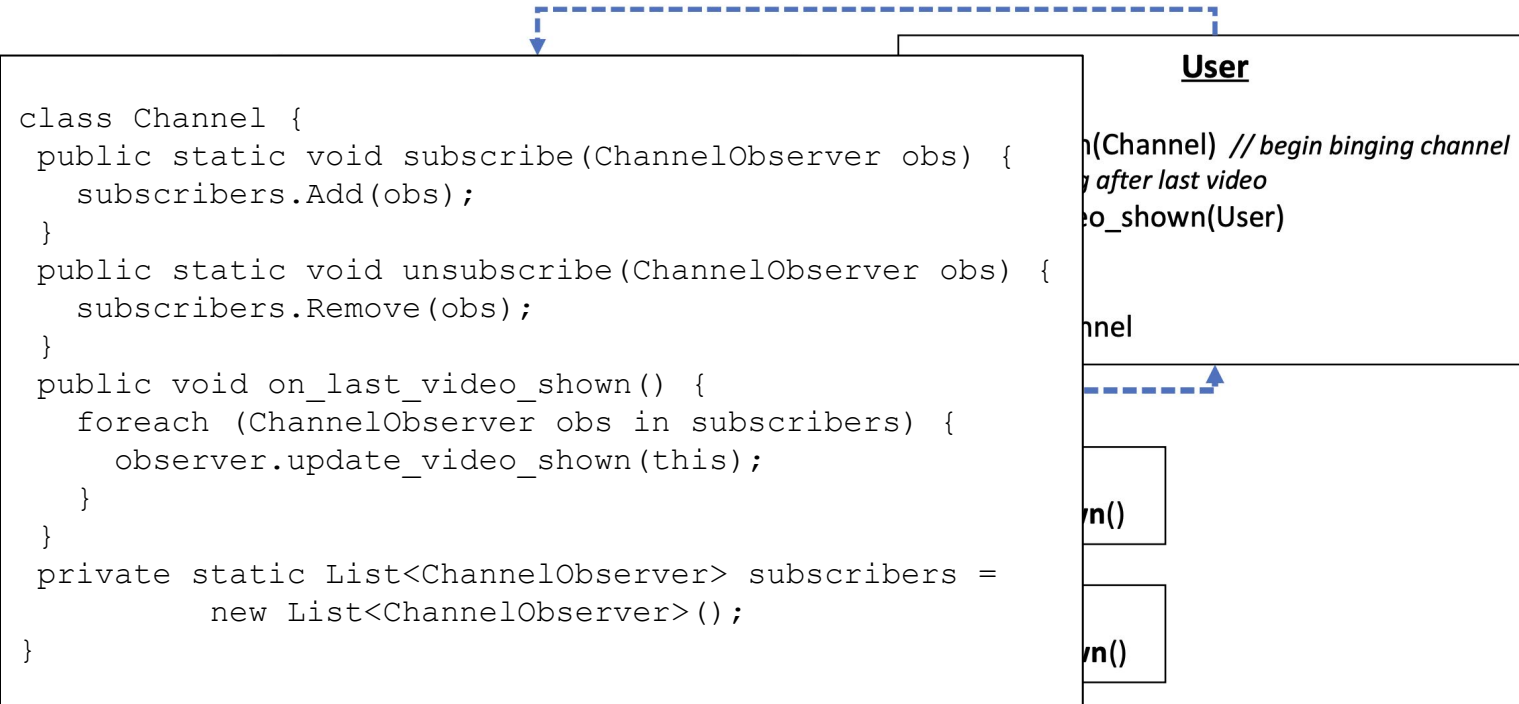
- The *observer pattern* (also called “*publish-subscribe*”) allows dependent objects to be notified automatically when the state of a subject changes. It defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified.



# Observer Pattern: bing-watch scenario

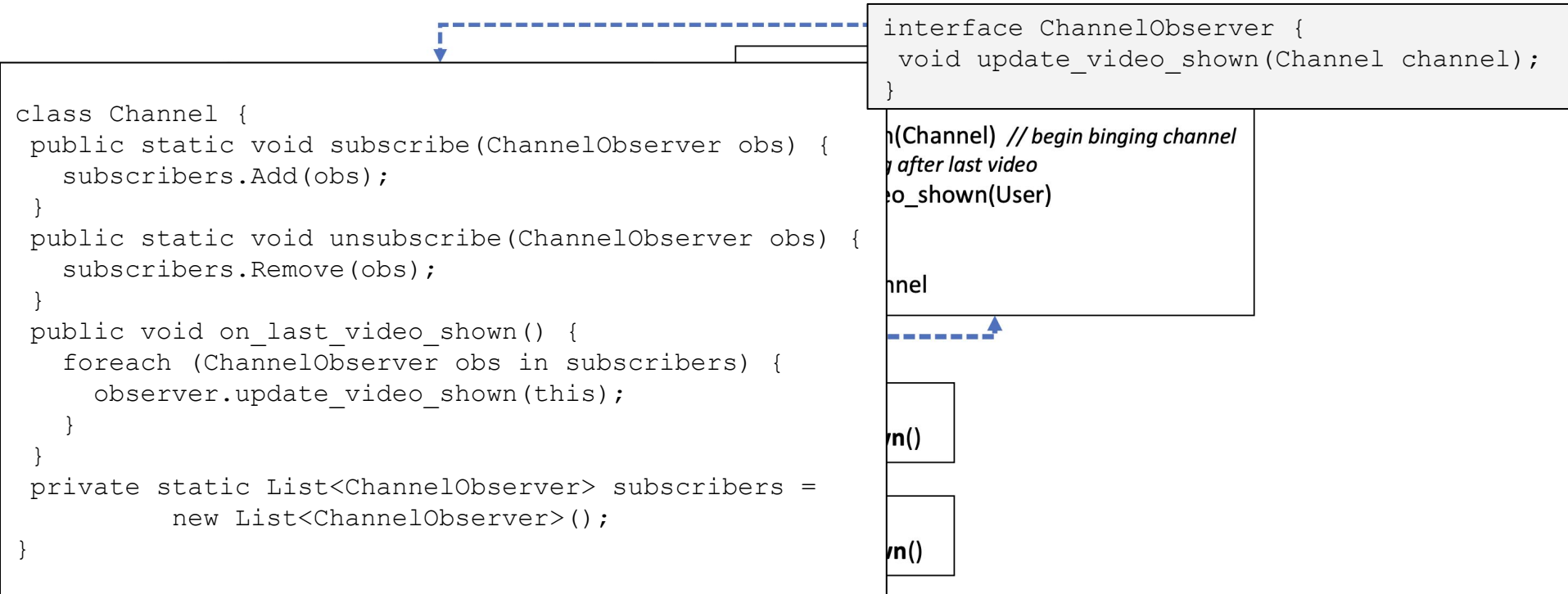


# Observer Pattern: bing-watch scenario





# Observer Pattern: bing-watch scenario



# Observer Pattern: bing-watch scenario



```
interface ChannelObserver {  
    void update_video_shown(Channel channel);  
}
```

```
on(Channel) // begin binging channel  
after last video  
video_shown(User)  
  
channel
```

```
class Channel {  
    public static void subscribe(ChannelObserver obs) {  
        subscribers.Add(obs);  
    }  
    public static void unsubscribe(ChannelObserver obs) {  
        subscribers.Remove(obs);  
    }  
    public void on_last_video_shown() {  
        foreach (ChannelObserver obs in subscribers) {  
            observer.update_video_shown(this);  
        }  
    }  
    private static List<ChannelObserver> subscribers =  
        new List<ChannelObserver>();  
}
```

```
class User: ChannelObserver {  
    public void update_video_shown(Channel c) {  
        if (c == binged_channel)  
            binged_channel = null;  
    }  
    public void binge_watch(Channel c) {  
        binged_channel = c;  
    }  
    private Channel binged_channel;  
}
```

# Observer Pattern: update functions

- Having multiple “update\_” functions, one for each type of state change, keeps messages **granular**

# Observer Pattern: update functions

- Having multiple “update\_” functions, one for each type of state change, keeps messages **granular**
  - Observers that do not care about a particular type of update can ignore it (via an empty implementation of the update function)

# Observer Pattern: update functions

- Having multiple “update\_” functions, one for each type of state change, keeps messages **granular**
  - Observers that do not care about a particular type of update can ignore it (via an empty implementation of the update function)
- Generally it is better to pass the newly-updated data as a parameter to the update function (**push**) as opposed to making observers fetch it each time (**pull**)

# Design patterns: takeaways

- Thinking about design before you start coding is usually worthwhile for large projects
  - Design around the most expensive parts of the software engineering process (usually maintenance!)
- Design patterns are re-usable solutions to common problems
- Be familiar with them enough to recognize when they're being used
  - and to know when to use them yourself
  - you can look up details of a pattern if you remember its name!
- Be mindful of and avoid common anti-patterns