

# Advanced Programming

ACSE-5: Lecture 10

Adriana Paluszny

# Overview: Where do we go from here?

- C++20
- Roles in programming teams
- Programming methodologies
- Agile
- Introduction to UML
- Design patterns

# C++20

- Before: C++ 14, C++ 17

## History [edit]

Changes applied to the C++20 working draft in July 2017 (Toronto) include:<sup>[7]</sup>

- [concepts](#) (what made it into the standard is a cut-down version; also described as "Concepts Lite"<sup>[8]</sup>)
- designated initializers
- `[=, this]` as a lambda capture
- template parameter lists on lambdas
- `std::make_shared` and `std::allocate_shared` for arrays

Changes applied to the C++20 working draft in the fall meeting in November 2017 (Albuquerque) include:<sup>[9][10]</sup>

- [three-way comparison](#) using the "spaceship operator", `operator <=>`
- initialization of an additional variable within a range-based `for` statement
- lambdas in unevaluated contexts
- default constructible and assignable stateless lambdas
- allow pack expansions in lambda *init-capture*
- string literals as template parameters
- atomic smart pointers (such as `std::atomic<shared_ptr<T>>` and `std::atomic<weak_ptr<T>>`)
- `std::to_address` to convert a pointer to a raw pointer

Changes applied to the C++20 working draft in March 2018 (Jacksonville) include:<sup>[11]</sup>

- removing the need for `typename` in certain circumstances

```
template<typename T>
concept EqualityComparable = requires(T a, T b) {
    { a == b } -> std::same_as<bool>;
    { a != b } -> std::same_as<bool>;
};
```

In [computer science](#), a **three-way comparison** takes two values A and B belonging to a type with a [total order](#) and determines whether A < B, A = B, or A > B in a single operation, in accordance with the mathematical [law of trichotomy](#).

## Library [edit]

- ranges (The One Ranges Proposal)<sup>[96]</sup>
- `std::make_shared` and `std::allocate_shared` for arrays<sup>[97]</sup>
- atomic smart pointers (such as `std::atomic<shared_ptr<T>>` and `std::atomic<weak_ptr<T>>`)<sup>[98]</sup>
- `std::to_address` to convert a pointer to a raw pointer<sup>[99]</sup>
- calendar and time-zone additions to `<chrono>`<sup>[100]</sup>
- `std::span`, providing a view to a contiguous array (analogous to `std::string_view` but `span` can mutate the referenced sequence)<sup>[101]</sup>
- `std::erase` and `std::erase_if`, simplifying element erasure for most standard containers<sup>[102]</sup>
- `<version>` header<sup>[103]</sup>
- `std::bit_cast<>` for type casting of object representations, with less verbosity than `memcpy()` and more ability to exploit compiler internals<sup>[104]</sup>
- feature test macros<sup>[105]</sup>

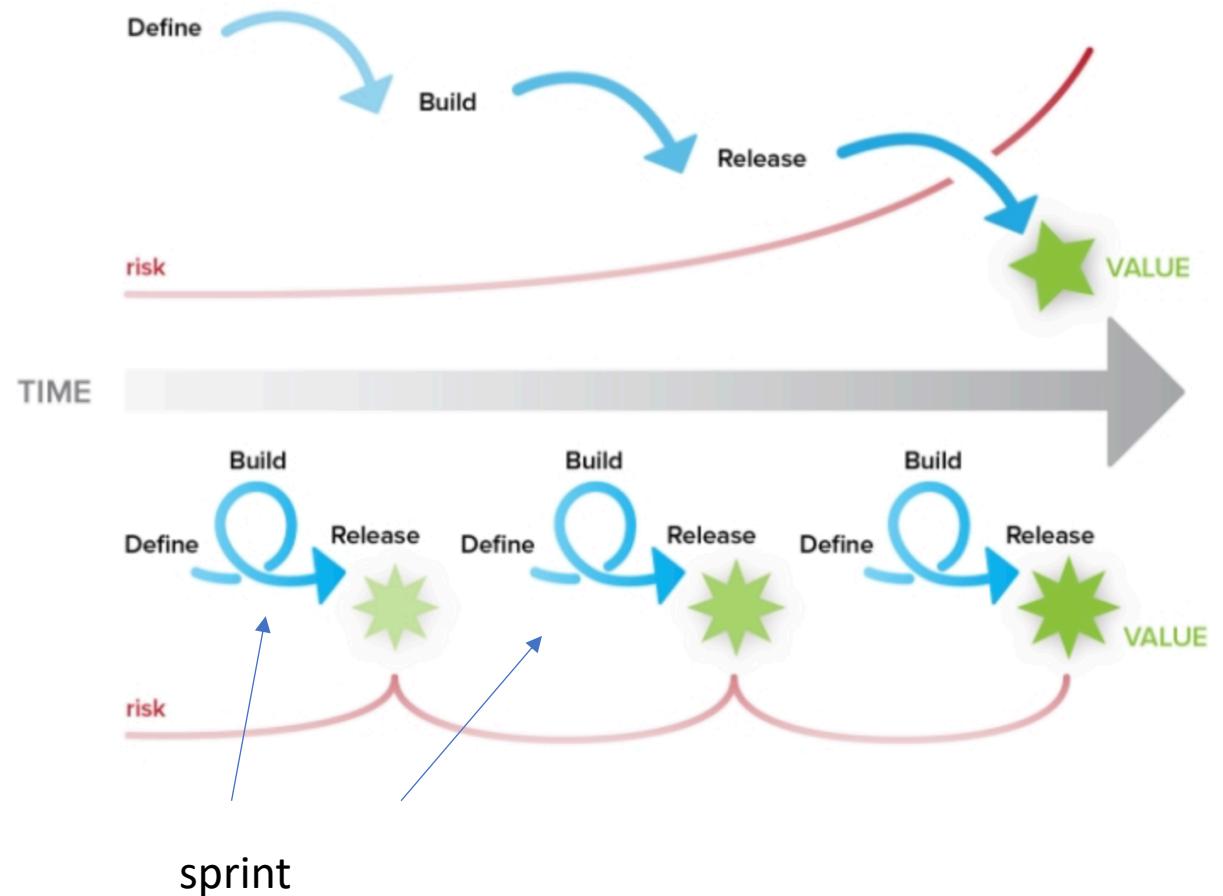
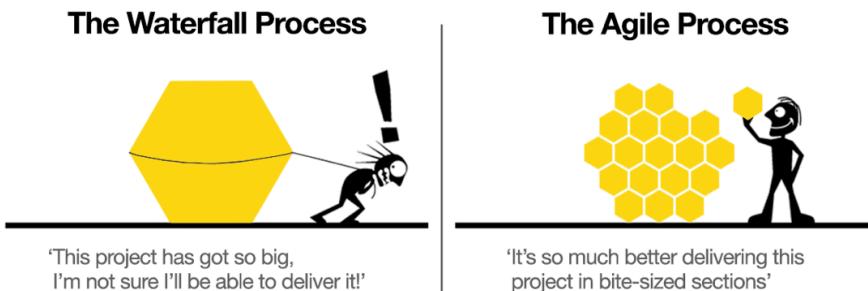
# Roles in programming teams

- Architect
- Tech Lead / CTO
- Software Developer Junior/Senior
- Scientific Developer
- Back end/ Front end Developer
- Full Stack Developer
- Product Manager/Lead
- Tester / Quality Control Manager



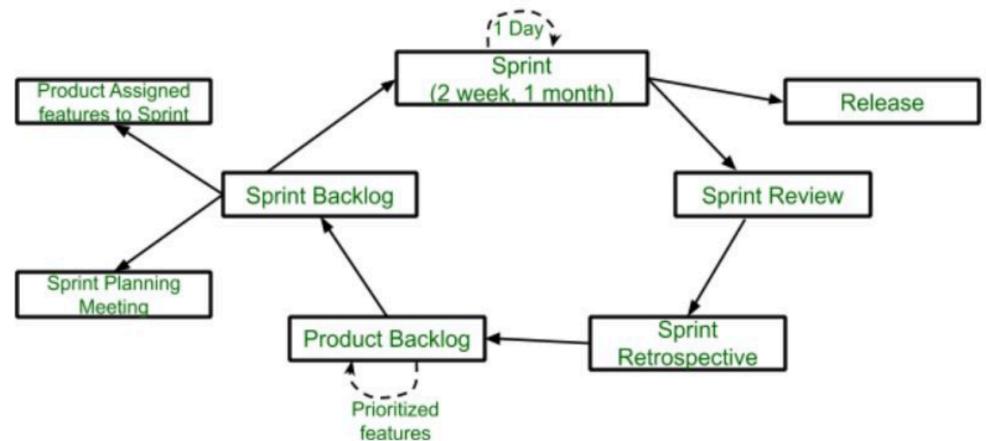
# Programming/Development methodologies

- Advanced programming for software engineering
- Waterfall, Prototype, Agile



# Agile

- Based on constant improvement (iterations/sprints)
- Iterative, dynamic, long vs. short nested cycles (30day/24hr cycles)



## Scrum

In Scrum framework, team work in iterations called Sprint which are 1-2 month long.

Scrum model do not allow changes in their timeline or their guidelines.

Scrum emphasizes self-organization.

In Scrum framework, team determines the sequence in which the product will be developed.

Scrum framework is not fully described. If you want to adopt it then you need to fill the framework with your own frameworks method like XP, DSDM or Kanban.

Scrum master: ensures the process but does not participate

## Extreme Programming (XP)

In Extreme Programming(XP), teamwork for 1-2 weeks only.

Extreme Programming allow changes in their set timelines.

Extreme Programming emphasizes strong engineering practices

In Extreme Programming, team have to follow a strict priority order or pre-determined priority order.

Extreme Programming(XP) can be directly applied to a team.

# (Micro) Introduction to UML

- Standard graphical notation for specifying, designing and documenting software
- Visualise design
- Increase understanding and communication of design within the team, stakeholders and customers
- Use case diagrammes (Behavioural)
- Class diagrammes (Structural)
- Sequence diagrammes (Behavioural)
- State diagrammes (Behavioural)
- Deployment diagramme (Structural)

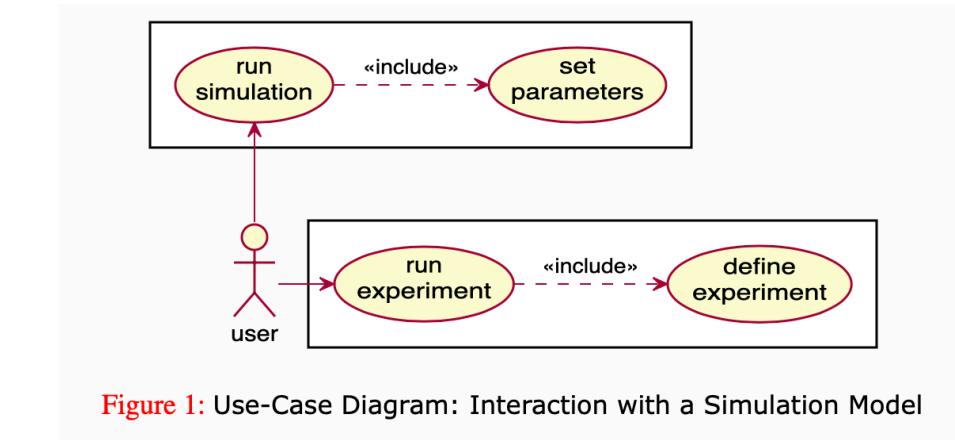
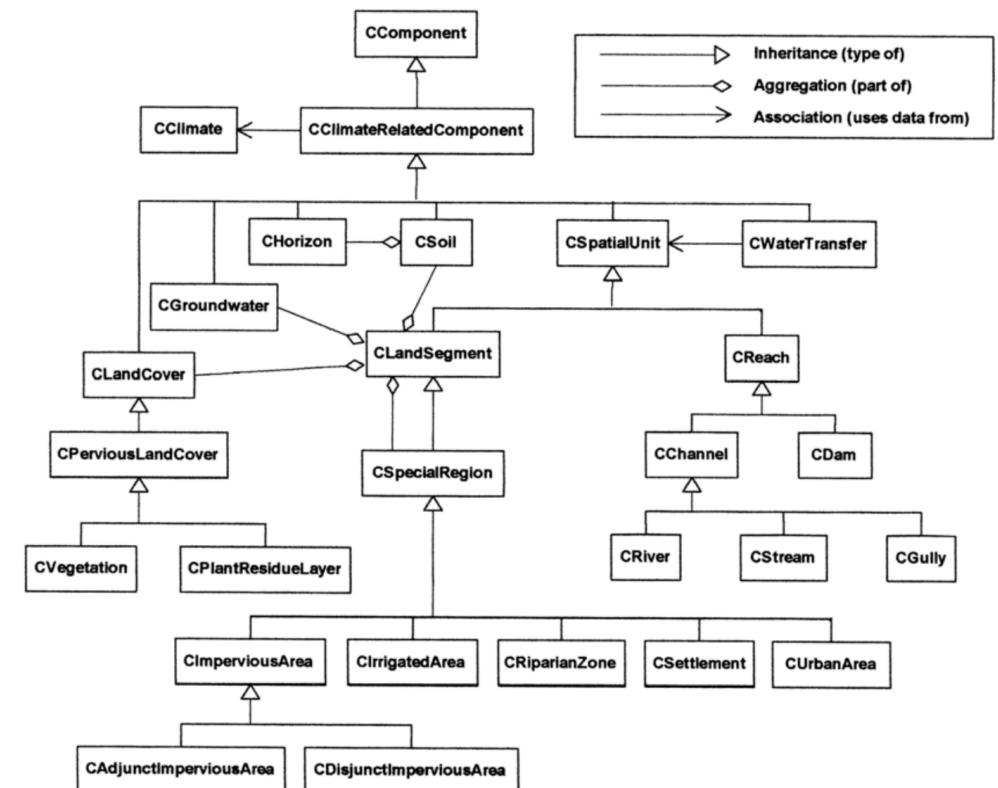


Figure 1: Use-Case Diagram: Interaction with a Simulation Model

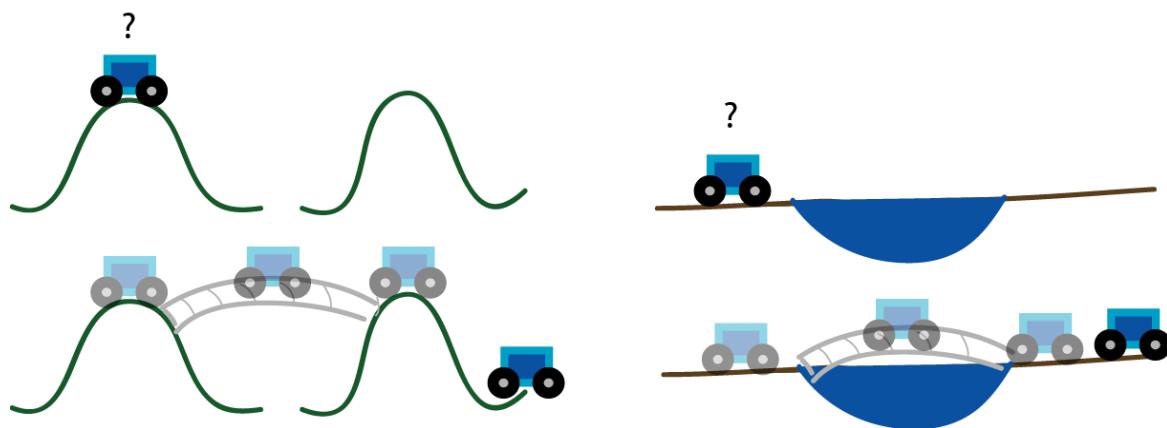


Further reading: <https://developer.ibm.com/technologies/web-development/articles/an-introduction-to-uml/>

UML class diagram showing some ACRU2000 Component classes and their relationships.

# What is a design pattern?

- describes **problem** that occurs over and over
- describes the core of the **solution** to the problem
- **consequences** of applying the pattern
- no two solutions will ever be the same



## Classification

- Purpose** (what the pattern does)
- Creational – creating objects
  - Structural – how objects are composed
  - Behavioral – interaction and responsibility distribution

- Scope** (application: classes or objects?)
- classes relationships: static / compile time
  - object relationships\*: dynamic / execution time

\* most patterns are in the object scope

# Examples of Design Patterns

Singleton: Creational Object

Bridge: Structural Object

Factory Method: Creational Class

Iterator: Behavioral Object

Visitor: Behavioral Object

Template: Behavioral Class

## The Catalog of C++ Examples

### Creational Patterns



#### Abstract Factory ★★★

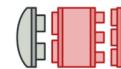
Lets you produce families of related objects without specifying their concrete classes.

[Main article](#)

[Code example](#)

[Usage in C++](#)

### Structural Patterns



#### Adapter ★★★

Allows objects with incompatible interfaces to collaborate.

[Main article](#)

[Conceptual example](#)

[Usage in C++](#)

[Multiple inheritance](#)



#### Builder ★★★

Lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code.

[Main article](#)

[Code example](#)

[Usage in C++](#)



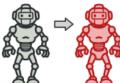
#### Bridge ★★★

Lets you split a large class or a set of closely related classes into two separate hierarchies—abstraction and implementation—which can be developed independently of each other.

[Main article](#)

[Code example](#)

[Usage in C++](#)



#### Prototype ★★★

Lets you copy existing objects without making your code dependent on their classes.

[Main article](#)

[Code example](#)

[Usage in C++](#)



#### Decorator ★★★

Lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors.

[Main article](#)

[Code example](#)

[Usage in C++](#)



#### Singleton ★★★

Lets you ensure that a class has only one instance, while providing a global access point to this instance.

[Main article](#)

[Naïve Singleton](#)

[Usage in C++](#)

[Thread-safe Singleton](#)



#### Facade ★★★

Provides a simplified interface to a library, a framework, or any other complex set of classes.

[Main article](#)

[Code example](#)

[Usage in C++](#)