# Programmierkurs
# Vorlesung 8
## Software Patterns

Andreas Naumann

Institut für Wissenschaftliches Rechnen
Universität Heidelberg

05. April 2023

## Software Patterns

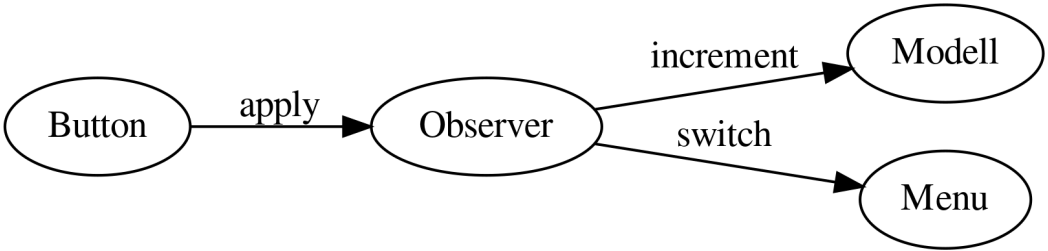
- Überblick
- Observer
- Singleton
- lazy (loading, initialization, evaluation)
- Adapter und Decorator
- Factory

# Software Patterns

- Wiederkehrende, ähnliche Probleme:
  - Bereitstellung von Daten
  - Speicherung (Kopie, Referenz)
  - Zuständigkeiten
  - Abhängigkeitsmanagement
- Wiederkehrende Grubstruktur von Lösungen in Software
- Ziele:
  - Wiederverwendbare Grundstruktur
  - Vereinfachung von Dokumentation durch Struktur
  - Ideen sprachunabhängig
  - Umsetzung/Möglichkeiten sprachabhängig

# Observer

- Grundproblem:
  - Eingaben/Änderungen beeinflussen andere Daten
  - Eingabe ist eigenständige Struktur
  - Ein Datum kann mehrere Ausgaben beeinflussen
- Häufig in GUIs:
  - Tabellenverwaltung: Abhängige Werte in Zellen
  - Menüs in Abhängigkeit von Inhalt/Zustand (z.B. speichern, compile)
- Ähnlich zu Model-view-controller:
  - Datenmodell informiert GUI (Konsistente Darstellung)
- Struktur
  - Änderung findet im Observer statt im Modell statt
  - Observer leitet weiter an Modell
  - und Beobachter




Beispiel

# Singleton

- Problemstellung:
  - Heterogene Initialisierung von Daten
  - globale Verfügbarkeit (z. B. `std::cout`)
- Struktur:
  - privater, statischer Pointer
  - versteckter Konstruktur
  - Zugriff auf Inhalte per statischer getter/setter Methoden
  - oftmals einmalige Initialisierung

# Singleton

```
struct Parameters
{
  static void init();
  static void set(std::string n,
      std::string v);
  static std::string get(std::string n,
      std::string v);
  private:
  Parameters() {}
  static std::unique_ptr<Parameters> data;
  std::map<std::string, std::string> content;
};
```

```
std::unique_ptr<Parameters> Parameters::data
void Parameters::init()
{
  if(!data) {
    data =
      std::make_unique<Parameters>();
  }
}
std::string Parameters::get(std::string n)
{
  if(!data) {
    throw NotInitialized();
  }
  auto it = data->content.find(n);
  if(it == data->end()) {
    throw NotFound(n);
  }
  return it->second;
}
```

# lazy (loading, initialization, evaluation)

- Problemstellung:
  - Zur Initialisierung liegen nicht alle Daten vor
  - Nicht alle Zwischenwerte/Objekte werden immer benötigt
  - Ausnutzen der Überlappung von teuren Operationen
- Struktur
  - klassische Konstruktoren oder Factories
  - `eval`, `init` Funktion ohne Argumente
  - oftmals mit caching

# lazy (loading, initialization, evaluation)

## Expression library

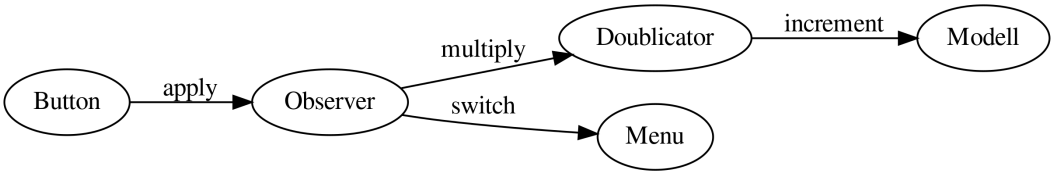
```
struct matrix;
struct vector;
struct mat_vec {
  const matrix& m;
  const vector& v;
  double get_entry(int r);
};
mat_vec operator*(const matrix& m,
  const vector& v)
{ return mat_vec(m, v); }
```

```
struct matrix {};
struct vector {
  vector(const mat_vec& mv)
  {
    data.resize(size(mv.m, 0));
    for(int r=0; r < size(mv.m,0); ++r)
    {
      data[r] = mv.get_entry(r);
    }
  }
  std::vector<double> data;
};
```

# Adapter und Decorator

- Problemstellung:
  - Vergleich von Bibliotheken mit verschiedenen Schnittstellen
  - Erweiterung von festen, vorgegebenen Schnittstellen
- Struktur:
  - Klasse `wrapper`, die die ursprüngliche Datenstruktur beinhaltet
  - Wrapper implementiert gewünschte Schnittstelle
  - Wrapper leitet Funktionsaufrufe an ursprüngliche Datenstruktur weiter
  - Mit Observer: Decorator ergänzt Argumente / Rückgabewerte

# Adapter und Decorator



# Factory

- Problemstellung:
  - Erzeugung einer Datenstrukture benötigt andere komplexe Datenstrukturen
  - Lange Abhängigkeitsketten/tiefe Bäume bei direkter Initialisierung
  - Bei optionalen Abhängigkeiten: Komplexe gegenseitige Abhängigkeiten zwischen Initialisierung und
  - Die Verwendung hat weniger/einfachere Abhängigkeiten
- Struktur:
  - Klasse `Factory`
  - statische Methoden zur Erzeugung/Initialisierung
  - Argumente der Methoden hängen nur von einfachen Strukturen ab

# Factory

```
struct Decorator {
  virtual int getInkrement() = 0;
};
struct DFactory {
  static std::shared_ptr<Decorator>
    getDecorator(const std::map<std::string, std::string>&);
};
```

# Zusammenfassung

- Pattern geben Struktur:
  - Speicherzuständigkeit (Singleton)
  - Separieren Aufgaben (MVC, Lazy, Factory)
- Vereinfachen Debugging
- **KEIN** Allheimittel
- oftmals keine scharfen Grenzen zwischen Patterns