LAZY ACQUISITION

EAGER ACQUISITION

DVA PŘÍSTUPY

LAZY

- Práci odkládáme, dokud to jde

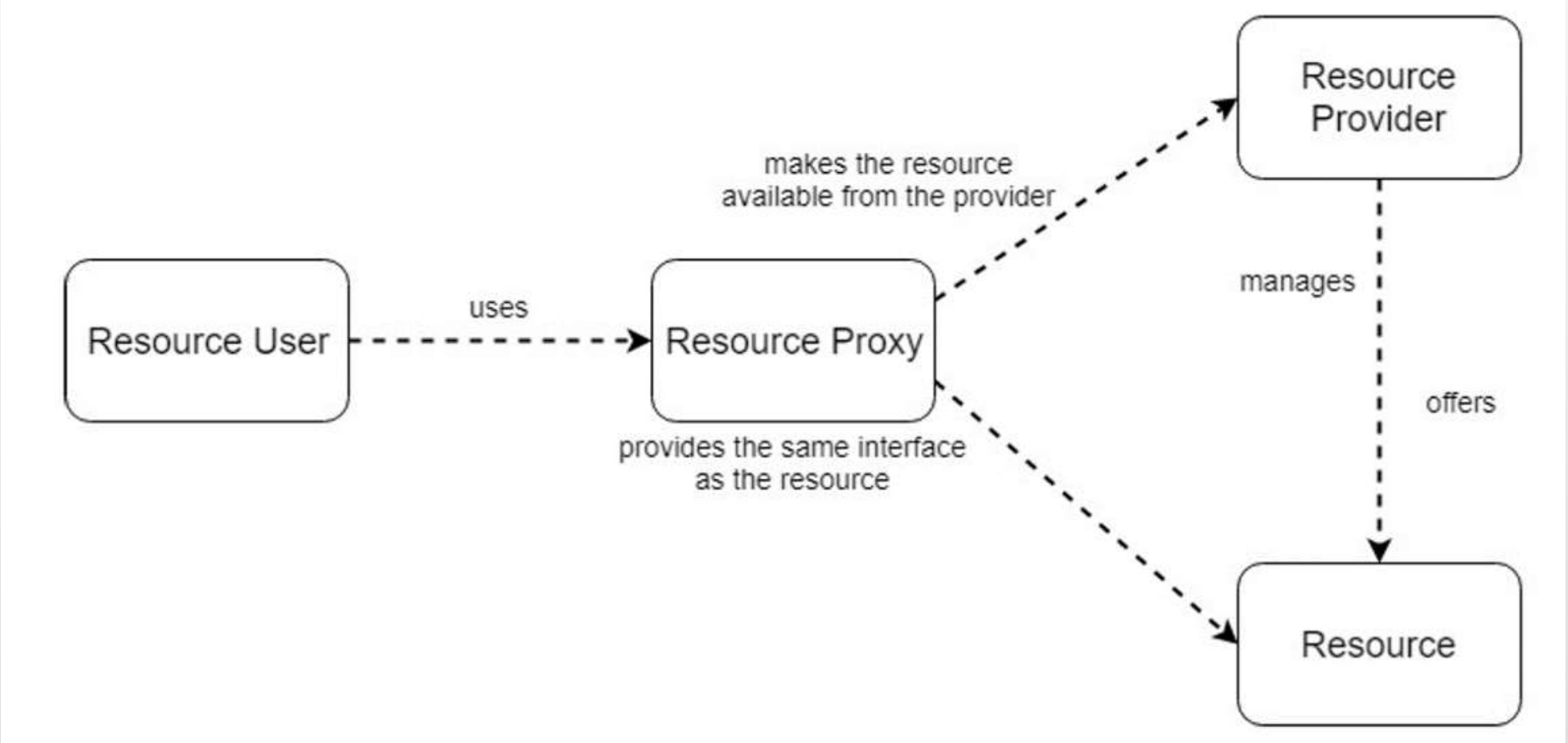
EAGER

- Práci provedeme co nejdřív

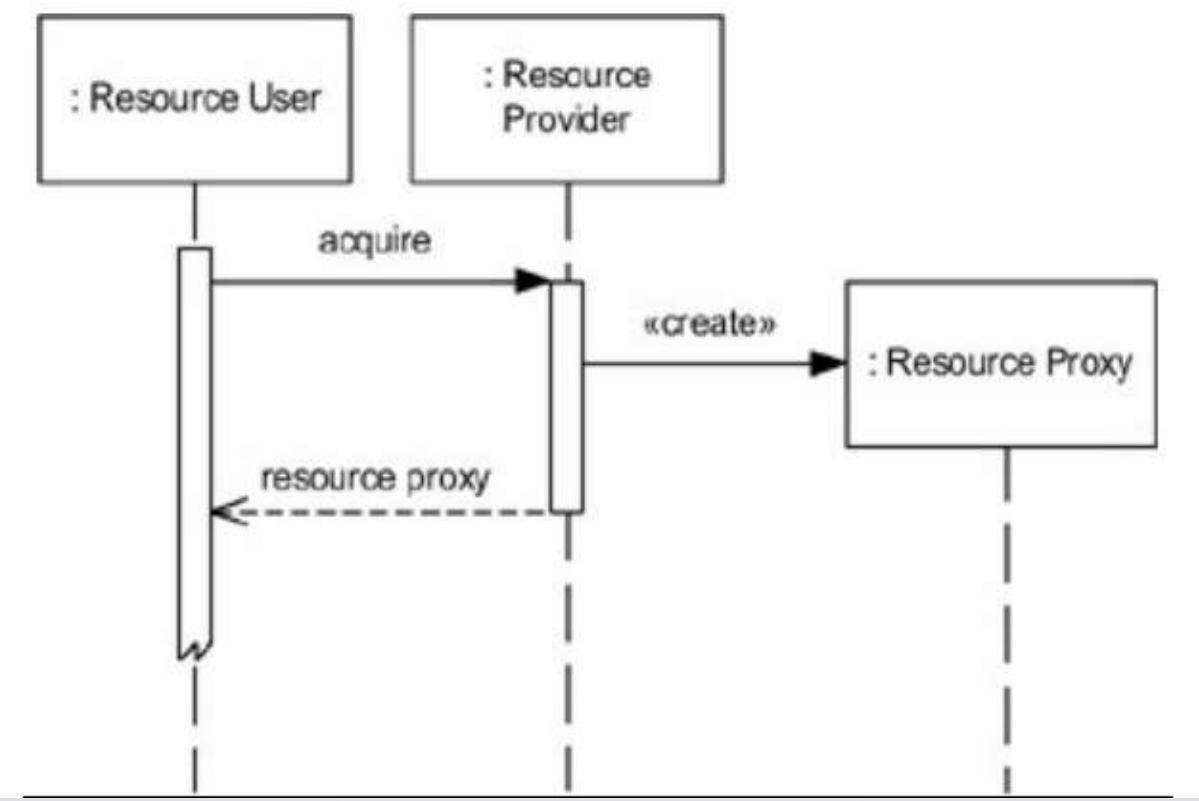
LAZY ACQUISITION - PŘÍKLAD

- Systém v nemocnici, který poskytuje přístup k datům
- Uloženy informace o jednotlivých pacientech (databáze, souborový systém, pojišťovna,…)
  - Základní informace
  - Záznamy o vyšetřeních
  - Snímky (rentgen,…)
- Data o pacientovi se načtou až o ně uživatel zažádá
  - Stačí nejdříve načíst jen ty podstatné, ne všechny snímky

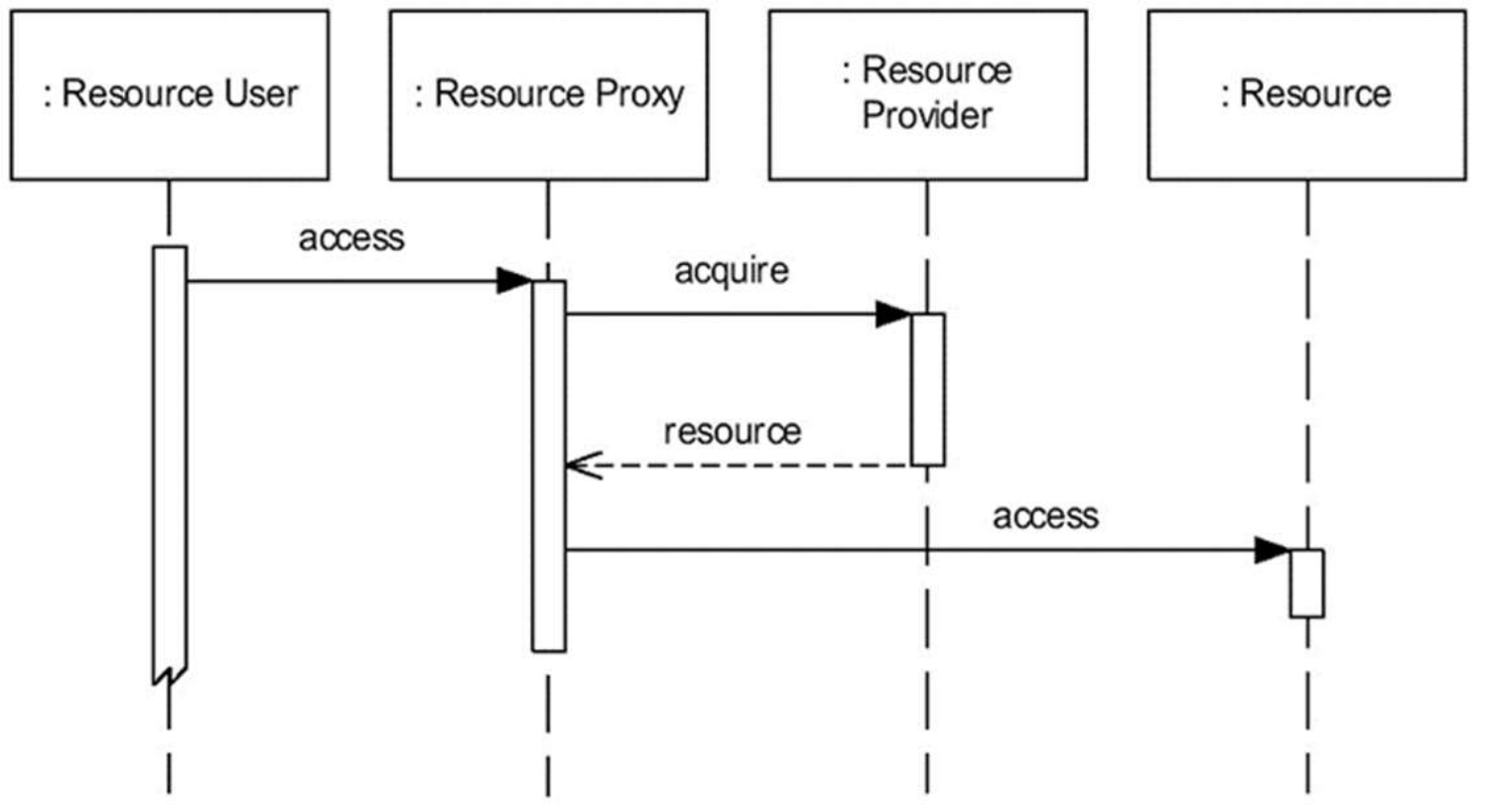
LAZY ACQUISITION - STRUKTURA



LAZY ACQUISITION – PRŮBĚH 1



LAZY ACQUISITION – PRŮBĚH 2



LAZY ACQUISITION –IMPLEMENTACE 1

```
interface IResource
    {
        public byte[] GetData();
    }

public class Resource : IResource
{
    public byte[] GetData()
    {
        // ...
    }
}

public class ResourceProvider
{
    private ResourceProxy rProxy;
    public Resource AccessResource(string ID)
    {
        // ...
    }

    public ResourceProxy GetProxy()
    {
        if(rProxy == null)
            rProxy = new ResourceProxy();
        return rProxy;
    }
}
```

```
public class ResourceProxy : IResource
{
    private Boolean acquired = false;
    private ResourceProvider rProvider = new ResourceProvider();
    private Resource resource;

    public byte[] GetData(string ID)
    {
        if (!acquired)
            AcquireResource(ID);
        return resource.GetData();
    }

    private void AcquireResource(string ID)
    {
        if (!acquired)
        {
            resource = rProvider.AccessResource(ID);
        }
        acquired = true;
    }

}
```

```
public class ResourceUser
{
    private ResourceProxy rProxy;
    private ResourceProvider rProvider = new ResourceProvider();
    public void AccessData(string ID)
    {
        if (rProxy == null)
            rProxy = rProvider.GetProxy();
        byte[] data = rProxy.GetData();
        //...
    }
}
```

LAZY ACQUISITION –IMPLEMENTACE 2

```
public class PatientManager {
  public static PatientRecord getPatientRecord(String patientId) {
    return dbWrapper.getRecordWithoutImages(patientId);
  }
}

interface PatientRecord {
  String getName();
  String getAddress();
  List getMedicalExams();
}

interface MedicalExam {
  Date getDate(); // Get the digitized image for this exam Image
  getImage();
}

interface Image {
  byte[] getData();
}
```

```
public class ImageProxy implements Image {
  ImageProxy(FileSystem aFileSystem, int anImageId) {
    fileSystem = aFileSystem;
    imageId = anImageId;
  }
  public byte[] getData() {
    if (data == null) {
        // Fetch the image lazily using the stored ID
        data = fileSystem.getImage(imageId);
    }
    return data;
  }
  byte data[];
  FileSystem fileSystem;
  int imageId;
}
```

LAZY ACQUISITION – (NE)VÝHODY

VÝHODY

- Dostupnost
- Stabilita
- Transparentnost
  - Uživatel nemusí vědět, jak přesně jsou prostředky získávány
- Rychlý start systému

NEVÝHODY

- Nepředvídatelnost
  - Doba načtení
- Pomalé získávání prostředků, když je uživatel potřebuje
- Paměť
  - Vytváření Resource Proxies

LAZY ACQUISITION - VYUŽITÍ

- Speciálnější návrhové vzory
  - Lazy instantiation
  - Lazy loading
  - Lazy evaluation
  - …
- V praxi
  - Haskell
    - Lazy evaluation – Spočítá/vyhodnotí toho jen tolik, kolik je potřeba k získání výsledku
  - Operační systémy
    - Lazy loading - Odloží načtení aplikačních knihoven, až když jsou potřeba

LAZY ACQUISITION – DALŠÍ SOUVISEJÍCÍ NÁVRHOVÉ VZORY

- Virtual Proxy
  - Resource Proxy
- Leasing, Evictor
  - Starají se o uvolnění prostředků
- Singleton
  - Instance singletonu se vytváří, až při prvním přístupu
- Eager Acquisition
  - Opačný přístup

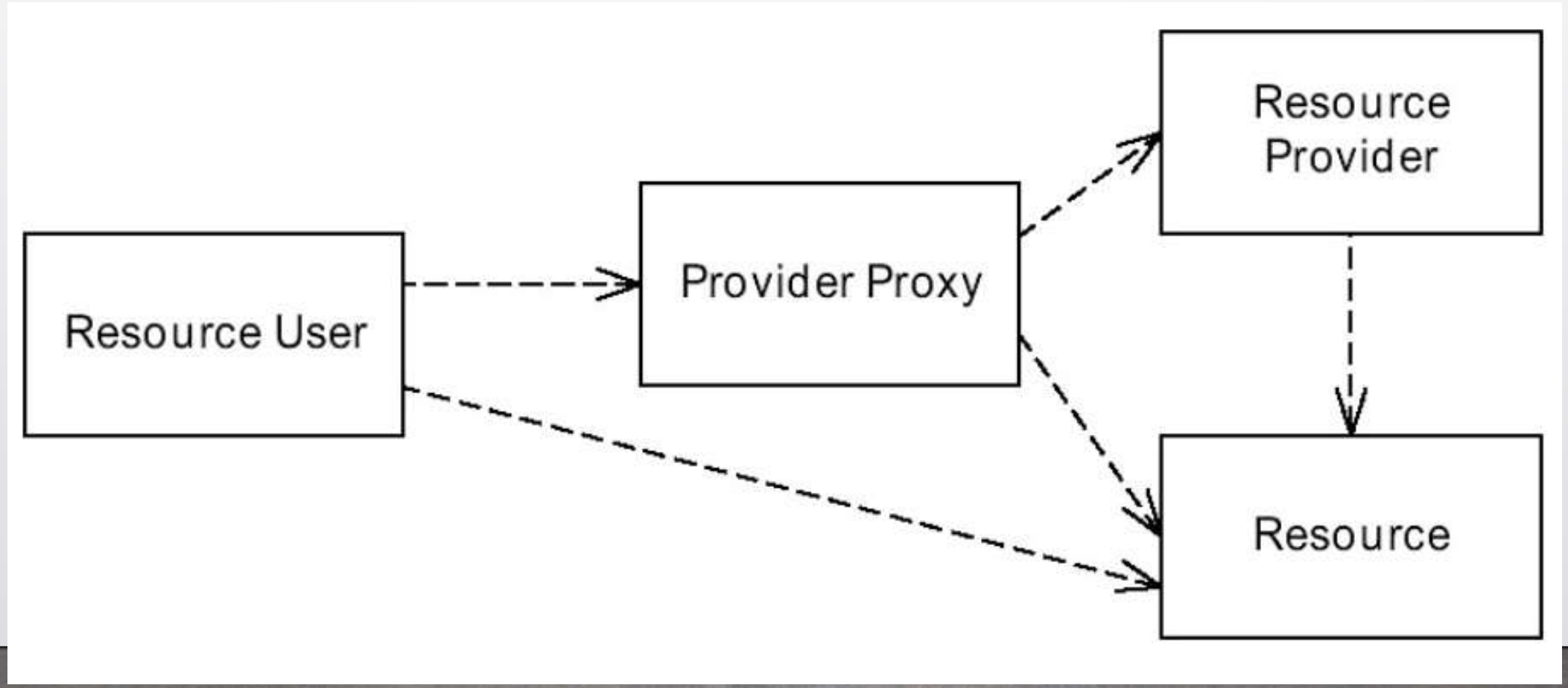
EAGER ACQUISITION - MOTIVACE

- GUI
  - Nutná rychlá odezva
  - Žádné „zamrznutí“ po uživatelově akci
  - Jak tomu zabránit?

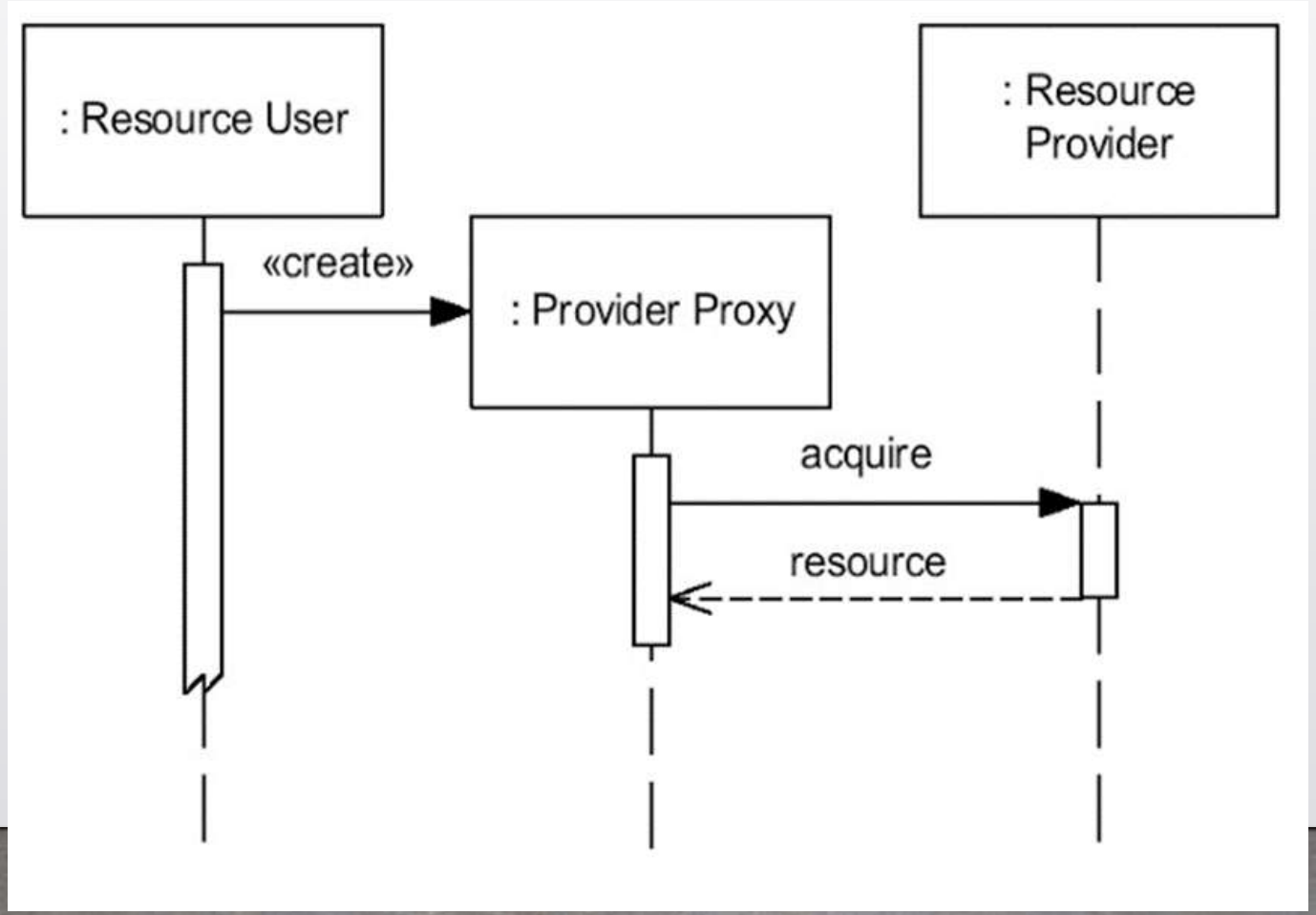
EAGER ACQUISITION

- Získáme všechny potřebné prostředky dříve, než je uživatel požaduje

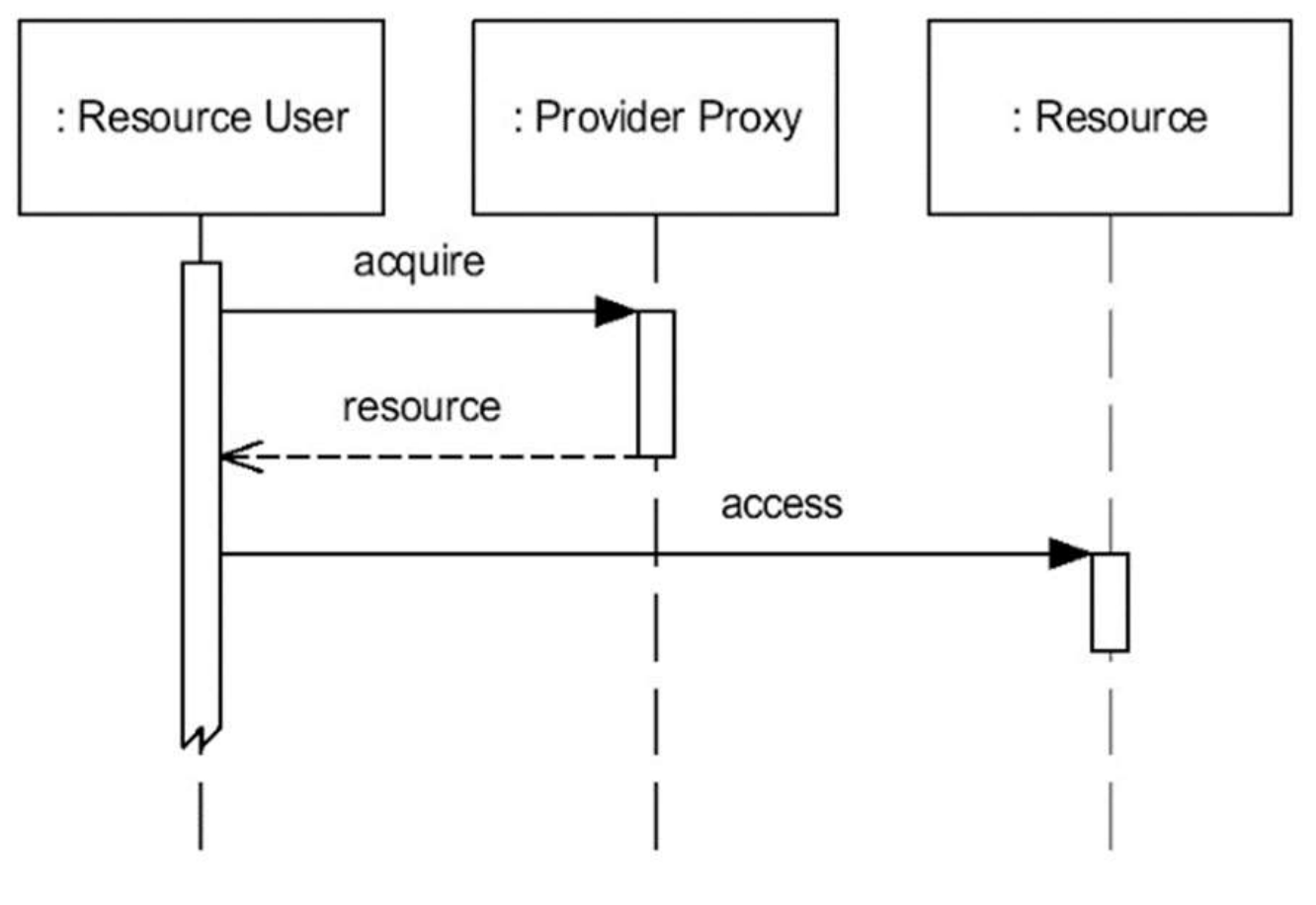
EAGER ACQUISITION – STRUKTURA



EAGER ACQUISITION – PRŮBĚH 1



EAGER ACQUISITION – PRŮBĚH 2



EAGER ACQUISITION – IMPLEMENTACE

```
public class ResourceProvider
{
    public Resource AccessResource(string ID)
    {
        // ...
    }
}
```

```
interface IResourceProvider
{
    public Resource AccessResource(string ID);
}

public class Resource
{
    public byte[] GetData()
    {
        // ...
    }
}

public class ResourceUser
{
    private ProviderProxy pProxy = new ProviderProxy();
    public void AccessData(string ID)
    {
        Resource r = pProxy.AccessResource(ID);
        byte[] data = r.GetData();
        //...
    }
}
```

```
public class ProviderProxy : IResourceProvider
{
    private ResourceProvider rProvider = new ResourceProvider();
    private Dictionary<string, Resource> resources = new Dictionary<string, Resource>();

    public ProviderProxy()
    {
        AcquireResources();
    }

    public Resource AccessResource(string ID)
    {
        return resources[ID];
    }

    private void AcquireResources()
    {
        while(/*Existuje zdroj k získání*/)
        {
            string resourceID = ...;
            Resource r = rProvider.AccessResource(resourceID);
            resources.Add(resourceID, r);
        }
    }
}
```

```cpp
const std::size_t block_size = 1024;
const std::size_t num_blocks = 32;

// Assume My_Struct is a complex data structure
struct My_Struct {
  int member;
    // ...
};

int main (int argc, char *argv[]) {
  try {
      Memory_Pool memory_pool(block_size, num_blocks);
      // ...
      My_Struct *my_struct =
          (My_Struct * ) memory_pool.acquire(sizeof(My_Struct));
      my_struct -> member = 42;
      // ....
  }
  catch (std::bad_alloc & ) {
      {
        std::cerr << "Error in allocating memory" << std::endl;
        return 1;
      }
      return 0;
  }
```

```cpp
class Memory_Pool {
  public: Memory_Pool(std::size_t block_size, std::size_t num_blocks):
    memory_block_(::operator new(block_size * num_blocks)),
    block_size_(block_size), num_blocks_(num_blocks) {
      for (std::size_t i = 0; i < num_blocks; i++) {
        void * block =
          static_cast < char * > (memory_block_) + i * block_size;
        free_list_.push_back(block);
      }
    }
  void * acquire(size_t size) {
      if (size > block_size_ || free_list_.empty()) {
          // if attempts are made to acquire blocks larger
          // than the supported size, or the pool is exhausted,
          // throw bad_alloc throw
          std::bad_alloc();
      } else {
          void * acquired_block = free_list_.front();
          free_list_.pop_front();
          return acquired_block;
      }
  }
  void release(void * block) {
    free_list_.push_back(block);
  }
  private:
    void * memory_block_;
    std::size_t block_size_;
    std::size_t num_blocks_;
    std::list < void * > free_list_;
};
```

EAGER ACQUISITION – (NE)VÝHODY

VÝHODY

- Předvídatelnost
- Výkon
- Transparentnost pro uživatele
- Flexibilita
  - Nenáročná změna strategie získávání prostředků

NEVÝHODY

- Nutnost spravování prostředků
- Statická konfigurace
  - Počet prostředků musí být odhadnutý dopředu
- Paměť
  - Větší množství dat, některé nejsou potřeba
- Pomalý start systému

EAGER ACQUISITION - VYUŽITÍ

- Speciálnější návrhové vzory
  - Eager instantiation
  - Eager loading
- Křeček
- V praxi
  - Memory Pool
  - Thread Pool
    - Vlákna se vytvoří dříve, než jsou potřeba
  - Ahead of time compilation
    - Java Virtual Machine

EAGER ACQUISITION – DALŠÍ SOUVISEJÍCÍ NÁVRHOVÉ VZORY

- Virtual Proxy
  - Provider proxy
- Abstract Factory
  - Resource User může použít k vytváření Provider proxies
- Lazy Acquisition
  - Opačný přístup

Závěrem

- Pro jednotlivé prostředky si musíme určit, který z návrhových vzorů nám více přispěje k:
  - Celkové dostupnosti prostředků
  - Stabilitě a předvídatelnosti systému
  - Rychlejšímu startu systému
- Dva zmíněné návrhové vzory jsou extrémy
- Lazy Acquisition - Nepotřebuji většinu prostředků, prostředky nejsou používány ihned,…
- Eager Acquisition – Potřebuji prostředky co nejrychleji, nevadí mi pomalejší start, …