ソフトウェア構築の原則：オブジェクト、設計、並行処理 (第2部：（サブ）システムの設計)

責任の分担

ジョナサン・アルドリッチ　チャーリー・ギャロッ

ド

コンピュー

ターサイエンス

学部

15-214

学習目標

- GRASP パターンを適用して
  設計の責任を割り当てる
- デザイン間のトレードオフを推論する

今日のトピック

- オブジェクト指向設計：”要件を特定し、ドメインモデルを作成した後、ソフトウェアクラスにメソッドを追加し、要件を満たすためにオブジェクト間のメッセージングを定義する。”
- でも、どうやって?

- コンセプトはクラスによってどのように実装されるべきか?
- どのメソッドがどこに属するのか?
- オブジェクトはどのように相互作用するべきか?
- これは非常に重要で、重要で、自明なことではない。

責任

- 責任とは、あるオブジェクトの行動上の義務に関するものである。
- 責任は2種類ある:
  - ご存知
  - している
- オブジェクトの責任には以下が含まれる:
  - オブジェクトの作成や計算など、何かをすることそのもの。
  - 他のオブジェクトのアクションを開始する
  - 他のオブジェクトの活動を制御・調整する
- オブジェクトの責任を知るには、以下のようなものがある:
  - カプセル化されたプライベート・データを知る

- かんれんちしき
- 導き出せるもの、計算できるものについて知っている

設計目標、原則、パターン

- 設計目標
  - 変化のためのデザイン、理解、再利用、分業、…。
- 設計原理
  - 低結合、高結合
  - 低い代表性のギャップ
  - デメテルの法則
- デザイン・ヒューリスティック（GRASP）
  - 情報専門家
  - クリエイター

– コントローラー

5

目標、原則、ガイドライン

- 設計目標
  - ソフトウェアに求められる品質
  - コストと便益の経済学が原動力
  - 例：変化のためのデザイン、理解、再利用、…
- 設計原則
  - ソフトウェア設計のガイドライン
  - 1つ以上の設計目標をサポートする
  - 例情報隠蔽、低反復ギャップ、低結合、高結合、…
- デザイン・ヒューリスティック
  - **低レベルの**設計決定に関する経験則
  - デザイン原則、ひいてはデザイン目標を推進する
  - 例クリエーター、エキスパート、コントローラー
- デザインパターン
  - 繰り返し発生する設計上の問題に対する一般的な解決策
  - 設計目標を促進するが、複雑さが増したり、トレードオフを伴ったりする可能性がある。
  - 例デコレーター、ストラテジー、テンプレートメソッド

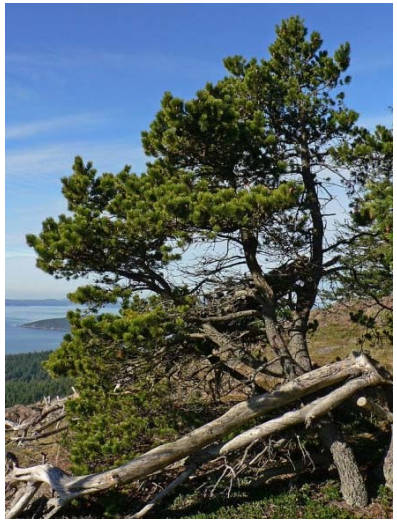
- 目標、原則、ヒューリスティック、パターンが衝突する可能性
  - プロジェクトの上位目標を解決する

GRASPパターン

- GRASP = 一般的な責任分担ソフトウェア・パターン
- **責任**分担のパターン
  - メソッドやフィールドをクラスに割り当てる際に、設計上のトレードオフを考慮する。
- GRASPパターンは、次のような**学習の助けと**なる。

– 本質的なオブジェクト設計を理解する
– 理路整然とした合理的で説明可能な方法で
  デザイン推論を適用する。
– より低レベルでローカルな推論
  デザインパターン

設計の原則:
低い表現ギャップ



問題スペース

ドメインモデル

オブジェクトと名前をイ

ンスパ

パインツリー
年齢 サイズ
収穫

n

1

フォレストソリューションスペース

オブジェクトモデル

レンジャー・エージェント
衛生（森林） サルベージ(森林)

デザイン
低い代表格差

- 各ドメインクラスに対してソフトウェアクラスを作成し、対応するリレーションシップを作成する。
- デザイン目標：変革のためのデザイン
- これは出発点に過ぎない！
  - すべてのドメイン・クラスがソフトウ

ェア対応を必要とするわけではなく、純粋なファブリケーションが必要な場合もある。

– 他の原則の方が重要な場合が多い

設計原理低カップリング

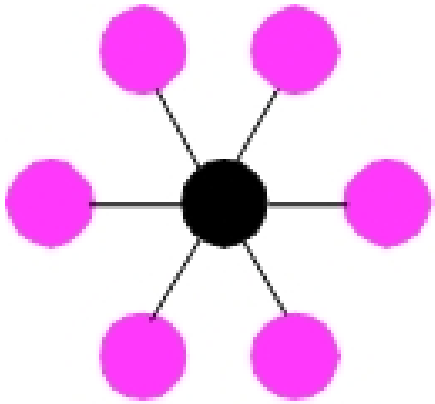
設計原理低結合

モジュールは、できるだけ他のモジュールに依存しないようにする。

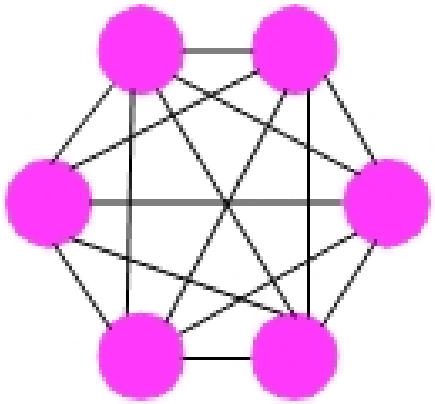
- 理解しやすさを高める（理解のためのデザイン）
  - 文脈の理解が乏しく、単独で理解する方が容易である。
- 変更のコストを削減する（変更のための設計）
  - 変更を加えるために必要な文脈はほとんどない

  - モジュールのインターフェイスが変更された場合、影響を受けるモジュールが少ない（波及効果の低減）
- 再利用の強化（再利用のための設計）
  - 依存関係が少なく、新しい状況に適応しやすい。

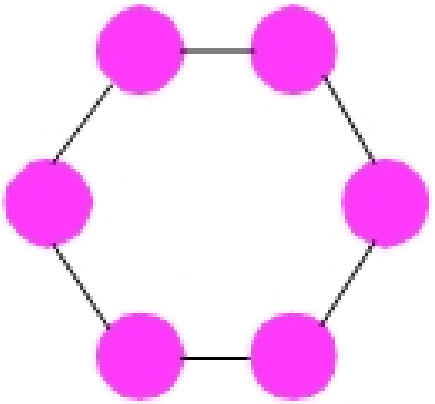
異なるカップリングを持つトポロジー



(A)



(B)



(C)

高カップリングは望ましくない

- 結合度の低い要素は、他のわずかな要素（クラス、サブシステム、...）にしか依存しない。
  - 「少数」は文脈に依存する
- 高いカップリングを持つクラスは、他の多くのクラスに依存している。
  - 関連するクラスが変化すると、局所的な変化を強いられる。局所的なクラスが変化すると、関連するクラスも変

化する（もろい、さざ波のような効果）。

– 単独で理解するのは難しい。
– 他の依存クラスを追加で存在させる必要があるため、再利用が難しい。
– 延長が難しい - 多くの場所で変更がある

どのクラスがカップリングされているのか?
カップリングはどうすれば改善できるのか?

```
クラス Shipment
{
        private List<Box> boxes;
        int getWeight() {。
                int w=0;
                for (ボックス box: boxes)
                        for (Item item: box.getItems())
                                w += item.weight;
                を返す;
}
クラス Box {
        private List<Item> items;
        Iterable<Item> getItems() { return items;}.
}
```

```
クラス アイテム
        箱入り;
        int weight;
}
```

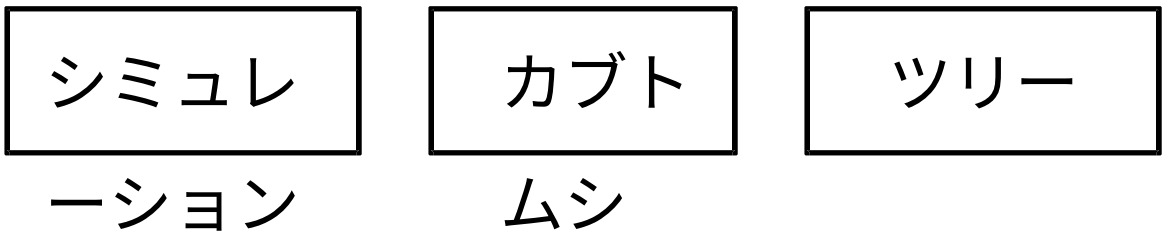
違うデザインだ。
カップリングはどうすれば改善できるのか?

```
クラス Box
{
        private List<Item> items;
        private Map<Item,Integer> weights;
        Iterable<Item> getItems() { return items;}.
        int getWeight(Item item) { return weights.get(item);}。
}
クラス アイテム
        private Box containedIn;
        int getWeight() { return containedIn.getWeight(this;}.
}
```

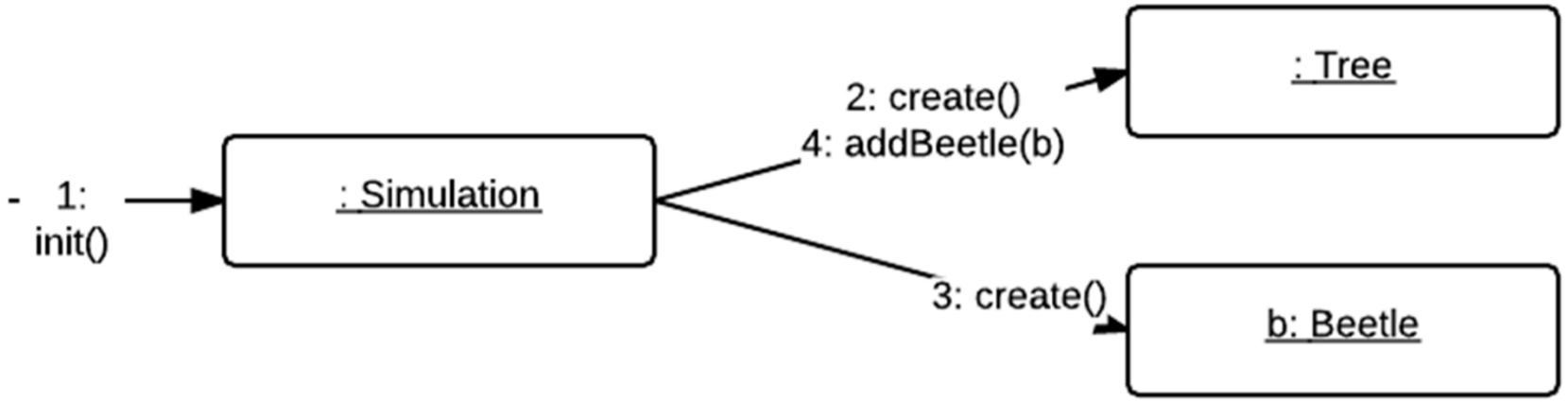
16

カップリング例

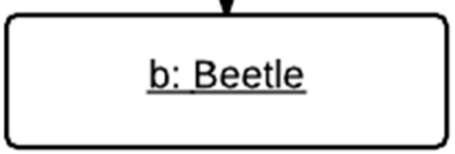
- ツリーを作り、カブトムシを "はびこらせる"



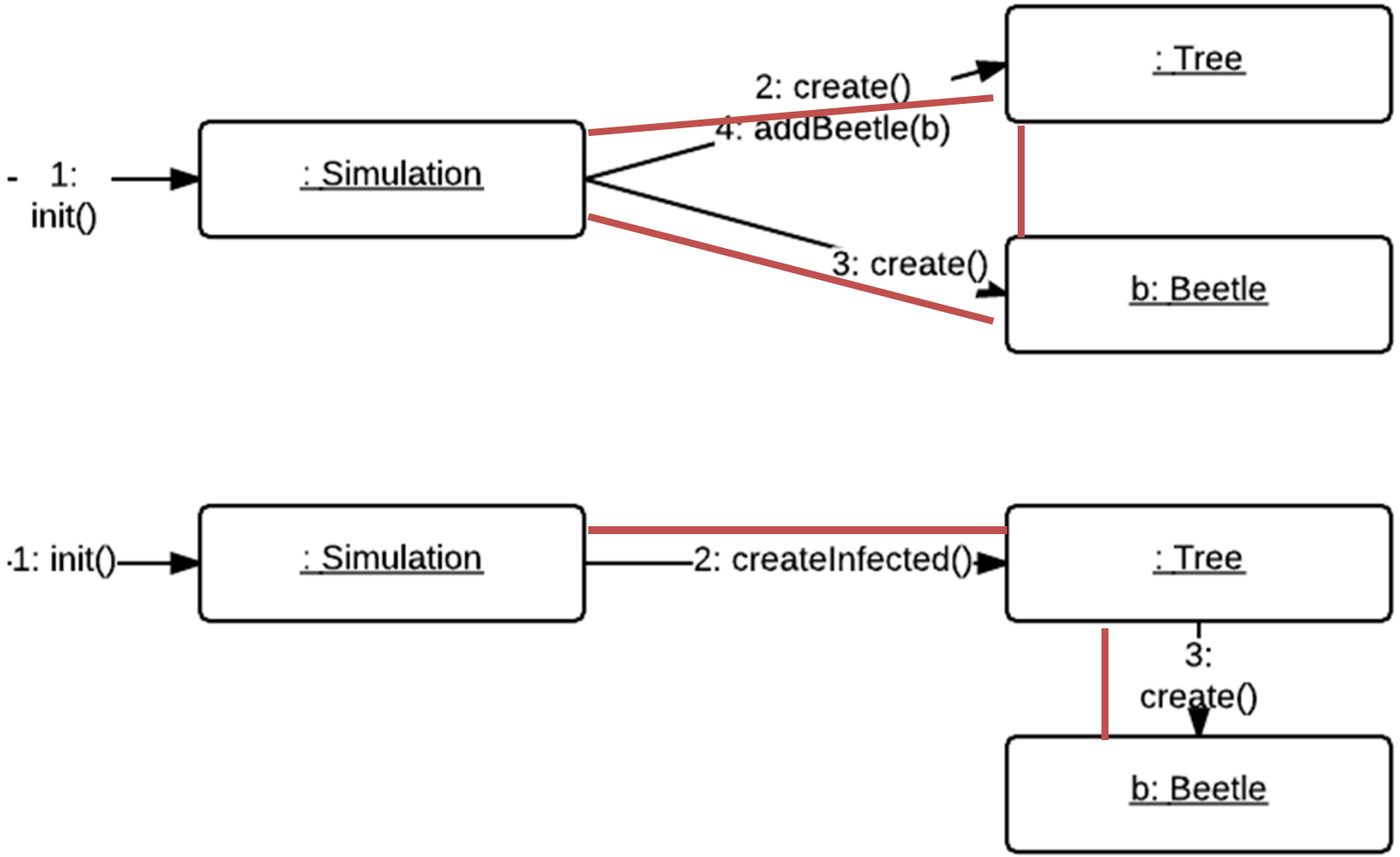
カップリング例


- 1:
init()
: Simulation
2: create()
4: addBeetle(b)
: Tree
3: create()
b: Beetle

カップリング例


b: Beetle

カップリング例



第二の解決策はカップリングが少ない

シミュレーションはビートルクラスを知らない

OOにおけるカップリングの一般的な形式
言語

- タイプXはタイプYのフィールドを持つ
- タイプXのメソッドmはタイプYを指す
  - 例：メソッドの引数、戻り値、ローカル変数、静的メソッド呼び出し
- タイプXはタイプYの直接的または間接的なサ

ブクラスである。

- タイプYはインターフェースであり、タイプXはそのインターフェースを実装している。

ロー・カップリングディスカッション

- 低結合は、すべての設計決定において留意すべき原則である。
- それは常に考え続けるべき根本的な目標である。
- これは、デザイナーがすべてのデザイン決定を評価する際に適用する評価原則である。
- 低カップリングは、より独立したクラスの設

計をサポートする。

- 文脈に依存する。結束や他の原則やパターンと合わせて考慮されるべきである。
- 実装への結合よりもインターフェースへの結合を優先する

デメテルの法則

- *各モジュールは、他のユニットに関する限られた知識しか持ってはならない。*
- 特に知らない人とは話さないこと!
- 例えば、a.getB().getC().foo()はない。

```
for (アイテム i:
        shipment.getBox().getItems())
        i.getWeight() ...
```

カップリングディスカッション

- サブクラスとスーパークラスの結合は特に強い
  - プロテクトされたフィールドとメソッドが見える
  - 例えば、メソッドのシグネチャの変更や抽象メソッドの追加などである。
  - *ガイドライン：カップリングを減らすために、継承よりもコンポジションを好む*
- 非常に安定した素子への高いカップリングは通常問題ない
  - 安定したインターフェースは変化しにくく、よく理解されている可能性が高い。

  - *実装への結合よりもインターフェースへの結合を優先する*
- カップリングは、数ある原則の中の1つである。
  - 結束力、低いレプリーグギャップ、その他の原則を考慮する

"規格外 "へのカップリング

- ライブラリやプラットフォームには、非標準の機能や拡張機能が含まれている場合があります。
- 例ブラウザ間のJavaScriptサポート
  - 古いコンテンツ 。

- JavaScriptでは...
  - MSIE: e1.innerText = "新しいコンテンツ"
  - Firefox: e1.textContent = "新しいコンテンツ"

設計目標

- 低カップリングがどのようにサポートされるかを説明する
  - 変化のためのデザイン
  - 理解しやすいデザイン
  - 分業設計
  - 再利用設計
  - …

設計目標

- 変化のためのデザイン
  - 他のオブジェクトへの依存が少なくなるため、変更が容易になる。
  - 変化が波及効果をもたらしにくい
- 理解しやすいデザイン
  - 理解すべき依存関係が少ない（例: a.getB().getC().foo()
- 分業設計
  - インターフェイスが小さく、分割しやすい

- 再利用設計
  - 複雑な依存関係なしに再利用しやすい

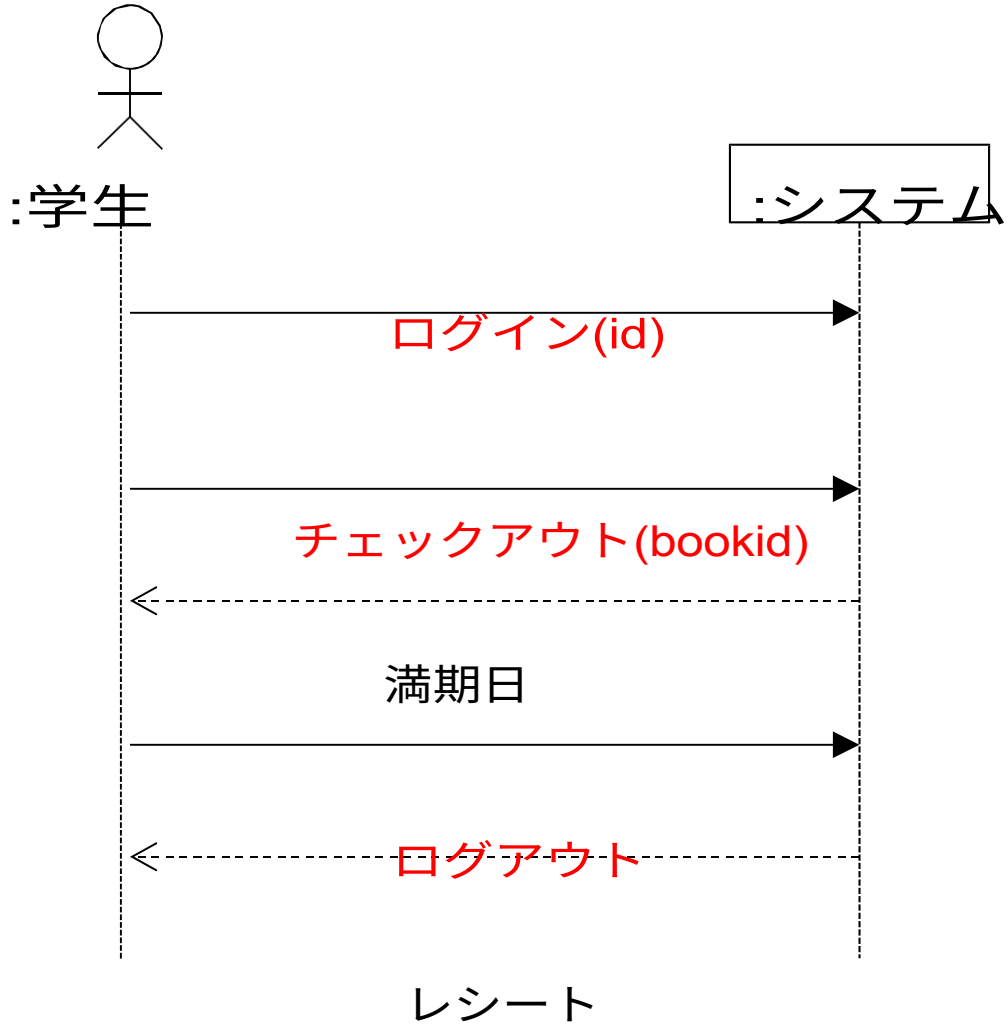
把握パターン：コントローラー デザインパターン：ファサード

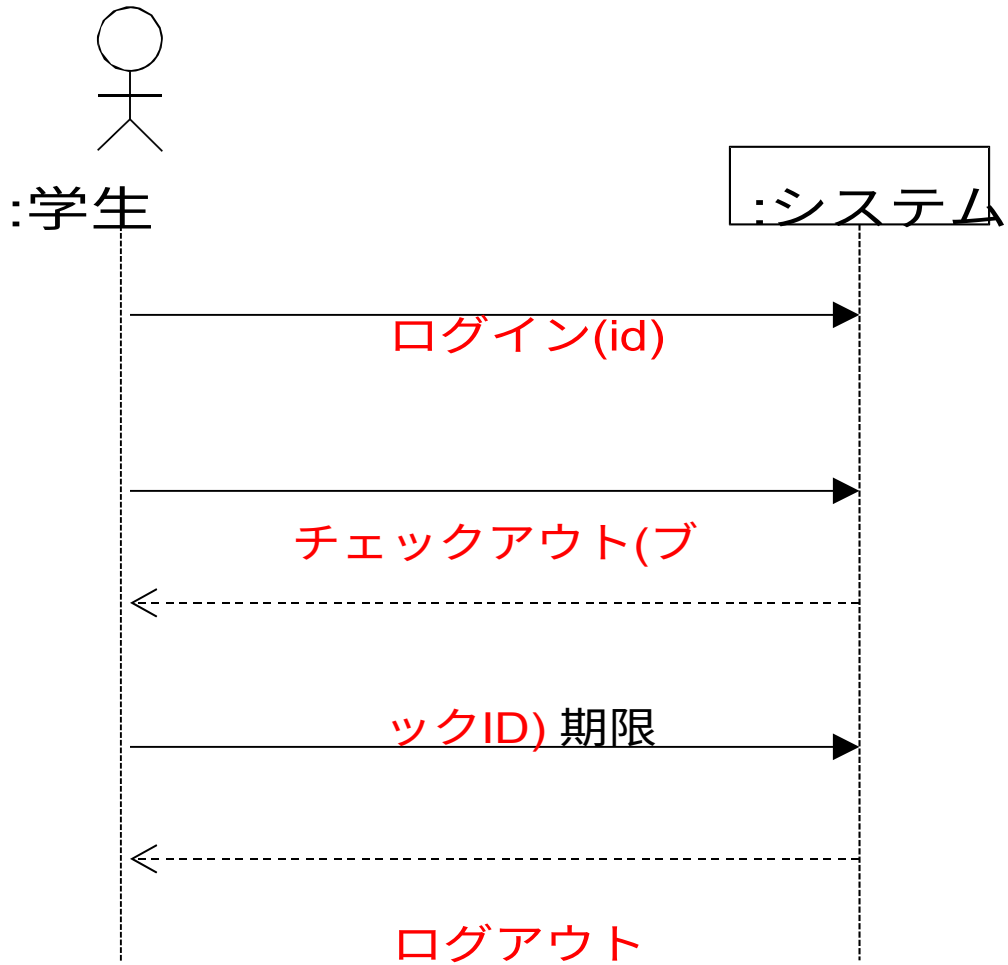
コントローラー（GRASP）

- 問題：システム操作（イベント）を受け取り、調整するオブジェクトは何か?
- 解決策を表すオブジェクトに責任を割り当てる。

– システム全体、デバイス、サブシステム（ファサードコントローラ）、または

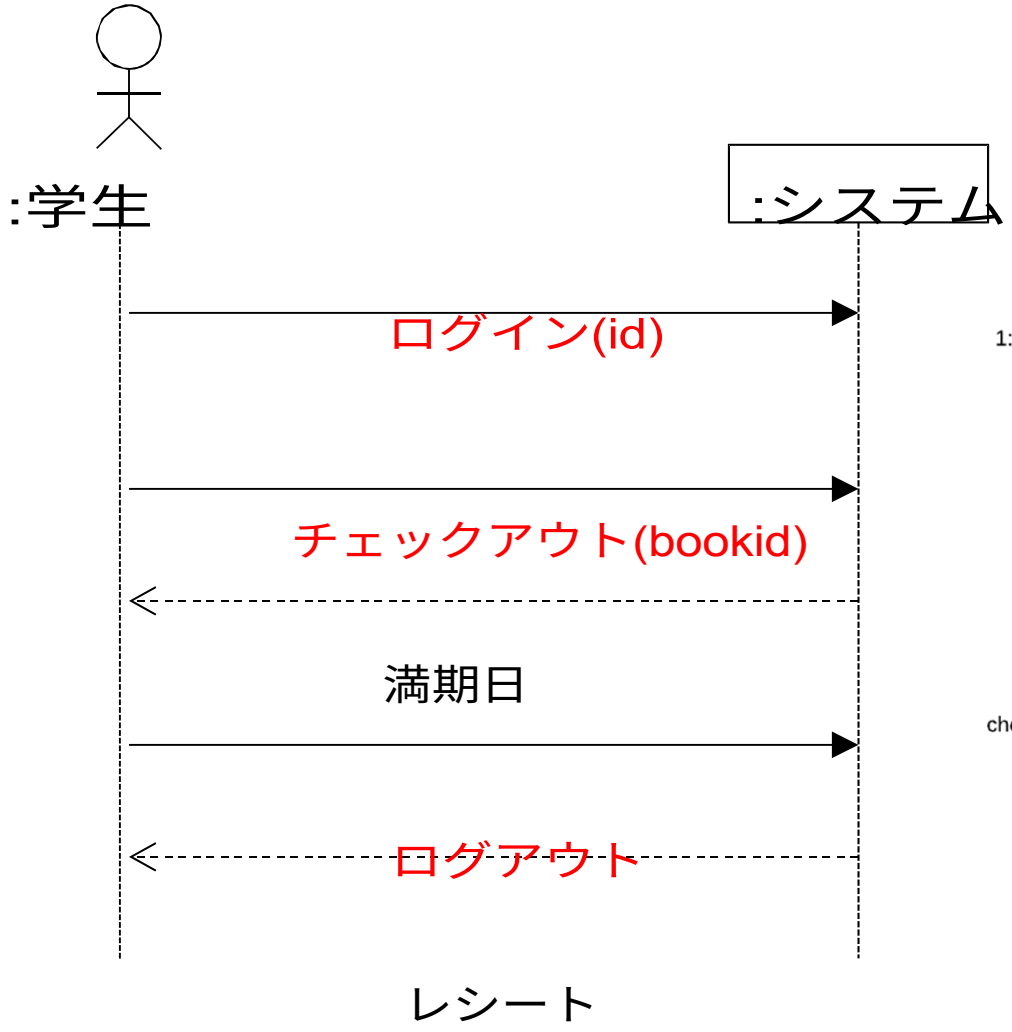
– システムイベントが発生するユースケースシナリオ（ユースケースコントローラ）

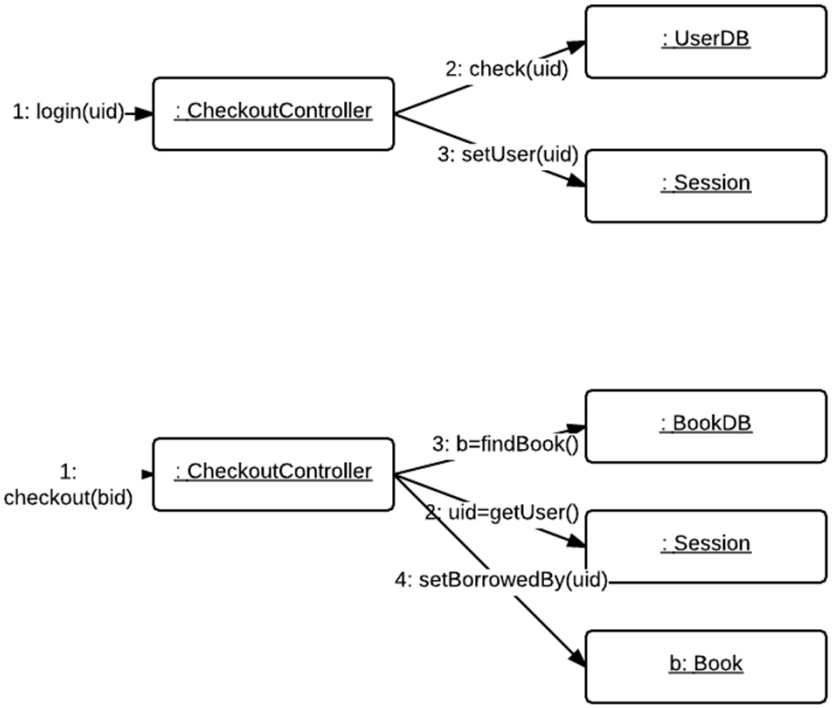

:学生
:システム
ログイン(id)
チェックアウト(bookid)
満期日
ログアウト
レシート



CheckoutController
ログイン(id: Int) checkout(bid: Int) logout()

レシート


:学生
:システム
ログイン(id)
チェックアウト(bookid)
満期日
ログアウト
レシート


: UserDB
2: check(uid)
1: login(uid)
: CheckoutController
3: setUser(uid)
: Session
: BookDB
3: b=findBook()
1:
checkout(bid)
: CheckoutController
2: uid=getUser()
: Session
4: setBorrowedBy(uid)
b: Book

コントローラーディスカッション

- コントローラーは調整役
  - それ自身はあまり働かない
  - 他のオブジェクトへの委譲
- ファサード・コントローラーは、システム・イベントが「多すぎない」場合に適している。
  - -> システム全体のコントローラーは1つ
- ファサードコントローラが過剰な責任で「肥大化」している場合に適したユースケースコントローラ（低凝集性、高結合性）
  - →特定のタスクのために複数の小さなコントローラー

- Façadeデザインパターンと密接な関係がある（今後の講義）

コントローラーデザインの議論
目標／戦略

- カップリングの減少
  - ユーザー・インターフェースとドメイン・ロジックは互いに切り離されている
    - 理解しやすさ：これらを単独で理解できる：
    - 進化性：UIとドメインロジックの両方が変更しやすい。
  - どちらも仲介役であるコントローラに結合しているが、
    この結合はそれほど有害ではない
    - コントローラーはより小型で安定したインターフェース
    - ドメインロジックの変更は、UIではなくコントローラに影響する。
    - ドメインロジックの設計を知らなくてもUIを変更できる
- 再利用のサポート
  - コントローラーはドメインロジックへのインターフェースとして機能する
  - より小さく、明示的なインターフェイスが進化可能性をサポー

トする

- しかし、肥大化したコントローラは結合を増やし、結束力を低下させる。

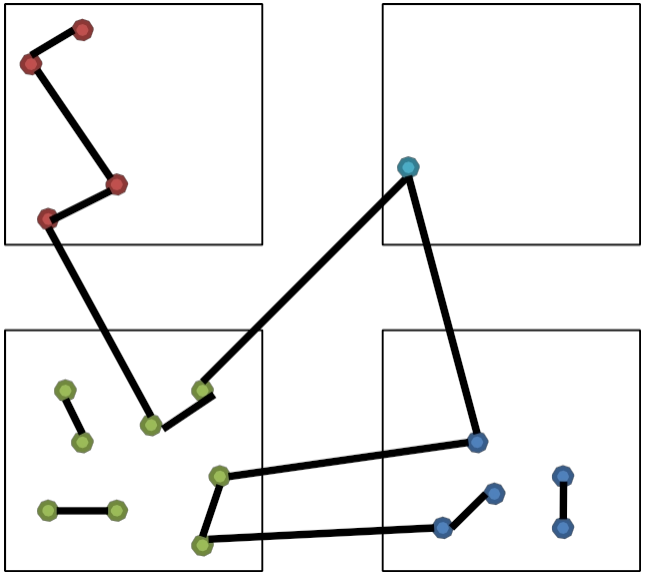
設計原則：高い結束力

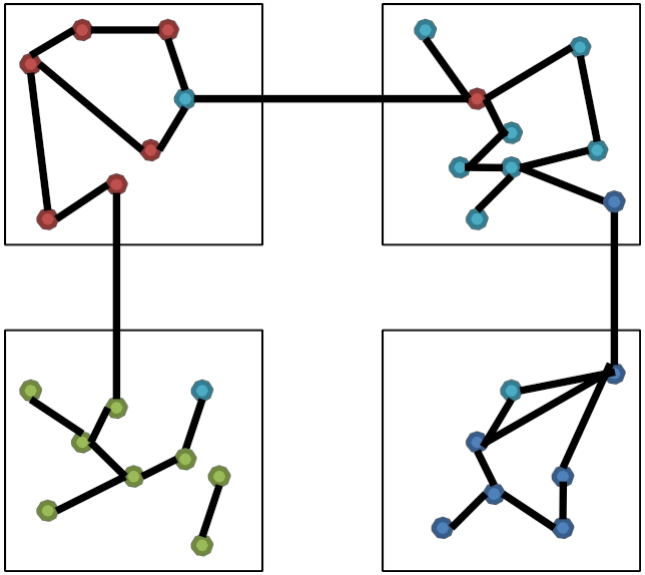
設計原則：結束

モジュールは、関連する責任の小さなセットを持つべきである。

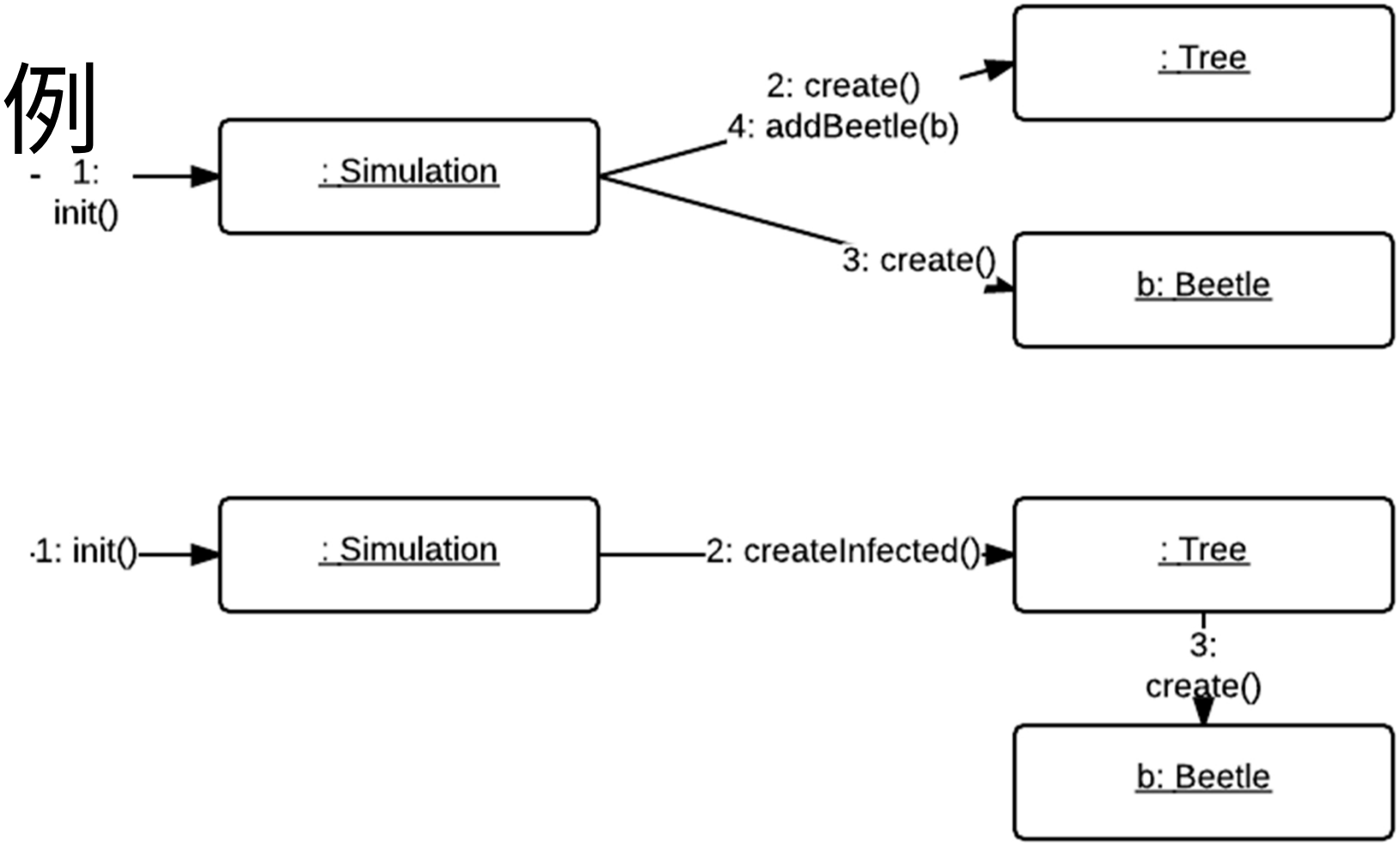
- わかりやすさの向上（わかりやすさのためのデザイン）
  - 小さな責任範囲の方が理解しやすい
- 再利用の強化（再利用のための設計）

– まとまりのある責任分担は、別のアプリケーションでも再現する可能性が高い。





シミュレーションにおける結束の例



登録責任

- 環境刺激に基づくシミュレーション・ステップのトリガー
- ドメインオブジェクトの作成を調整する

```
クラス DatabaseApplication
        //データベース・フィールド
        //...ロギング・ストリーム
        //...キャッシュ・ステータス
        public void authorizeOrder(Data data, User currentUser, ...){.
                // 認証をチェックする
                // 同期のためにオブジェクトをロックする
                // バッファの検証
                // 操作開始のログ
                // 操作を実行する
                // 操作終了のログ
                // オブジェクトのロックを解除する
        }
        public void startShipping(OtherData data, User currentUser, ...){.
                // 認証をチェックする
                // 同期のためにオブジェクトをロックする
                // バッファの検証
                // 操作開始のログ
                // 操作を実行する
                // 操作終了のログ
                // オブジェクトのロックを解除する
        }
}
```

グラフ実装における凝集性

```
クラス グラフ
        Node[] nodes;
        boolean[] isVisited;
}
クラス アルゴリズム {
        int shortestPath(Graph g, Node n, Node m) {
                for (int i; ...)
                        if (!g.isVisited[i]) {。
                                ...
                                g.isVisited[i] = true;
                        }
                }
                                                }
```

グラフはデータだけでなく、次のような任務も担っている。

アルゴリズムの責任

}

15-214 を返す;

モノポリーの例

結束力が高いのはどちらのデザインか?

```
クラス プレイヤー {
        取締役会
        /* in code somewhere... */ getSquare(n);
        Square getSquare(String name) {.
                for (正方形 s: board.getSquares())
                        if (s.getName().equals(name))
                                return s;
                nullを返す;
}}
クラス プレイヤー {
        取締役会
        /* どこかのコードで... */ board.getSquare(n);
}
クラス Board{
        リスト<正方形>の正方形;
```

```
正方形 getSquare(String name) {
        for (正方形 s: 正方形)
                if (s.getName().equals(name))
                        return s;
        nullを返す;
```

15-
}2}14

結束を見極めるヒント

- 1つのコンセプトにつき1色を使用
- そのコンセプトのすべてのコードを色
  で強調表示
- クラスやメソ
  ッドには色を
  付けるべき

ない。

結束を見極めるヒント

- 何が "コンセプト "なのか、明確な定義はない。
- コンセプトはより小さなコンセプトに分割できる
  - 検索付きグラフ vs. 基本グラフリズム vs. 基本グラフ＋検索フ

＋具体的検索アルゴリズム など

- 技術的判断を必要とする

結束ディスカッション

- 凝集性が非常に低い：あるクラスが、非常に異なる機能領域で多くのことを一手に担っている。
- 低い凝集性：あるクラスが、ある機能領域における複雑なタスクを一手に引き受けている。
- 高い結束力：クラスは1つの機能領域で中程度の責任を負い、タスクを遂行するために他のクラスと協力する。
- 高い結束力の利点
  - クラスの維持が容易になる
  - 理解しやすい
  - 多くの場合、低カップリングをサポート
  - きめ細かな責任感で再利用をサポート

- 経験則：凝集性の高いクラスは、関連性の高い機能のメソッドが比較的少ない。

結合 vs 凝集（極端なケース）

極端なケースを考えてみよう:

- カップリングが非常に低い?
- 非常に高い結束力?

```
クラス グラフ
        Node[] nodes;
        boolean[] isVisited
        ;
}
クラス アルゴリズム {
        int shortestPath(Graph g, Node n, Node m) {
                for (int i; ...)
                        if (!g.isVisited[i]) {。
                                …
```

```
                                        g.isVisited[i] = true;
                                    }
                                }
                                を返す;
    }
```

結合 vs 凝集（極端なケース）

- すべてのコードを1つのクラス/メソッドにまとめる
  - 結合度は非常に低いが、凝集度は非常に低い
- 各ステートメント
  - 凝集力は非常に高いが、カップリングは非常に高

い

- 良いトレードオフを見つける。

把握パターン：情報エキスパート

情報エキスパート (GRASPパターン/デザイン・ヒューリスティック)

- ヒューリスティック：**責任を果たすために必要な情報を持っているクラスに責任を割り当てる**
- 責任を明確にすることで、責任の分担を始める！

- 一般的には直感に従う
- ドメイン・モデル・クラスの代わりにソフトウェア・クラス
  - ソフトウェアクラスがまだ存在しない場合は、ドメインモデルで適切な抽象化を探す（→対応）。

貨物の重量を計算するためのすべての情報を持っているクラスはどれか?

```
クラス Shipment {
        private List<Box> boxes;
        int getWeight() {。
                int w=0;
                for (ボックス box: boxes)
                        for (Item item: box.getItems())
                                w += item.weight;
                を返す;
}
クラス Box {
        private List<Item> items;
        Iterable<Item> getItems() { return items;}.
```

```
}
クラス アイテム
        箱入り;
        int weight;
}
```

情報の専門家→「Do It Myself 戦略"

- 専門家は通常、ソフトウェア・オブジェクトが、それが表現する現実世界の無生物に対して通常行われる操作を行うような設計を導き出す。
  - セールスはその総額を教えてくれない。
- OOデザインでは、すべてのソフトウェア・オブジェクトは"生きている"、あるいは"アニメーションしている"。

- 彼らは自分が知っている情報に関連したことをする。

把持パターン：クリエイター

クリエイター (GRASPパターン/デザイン・ヒューリスティック)

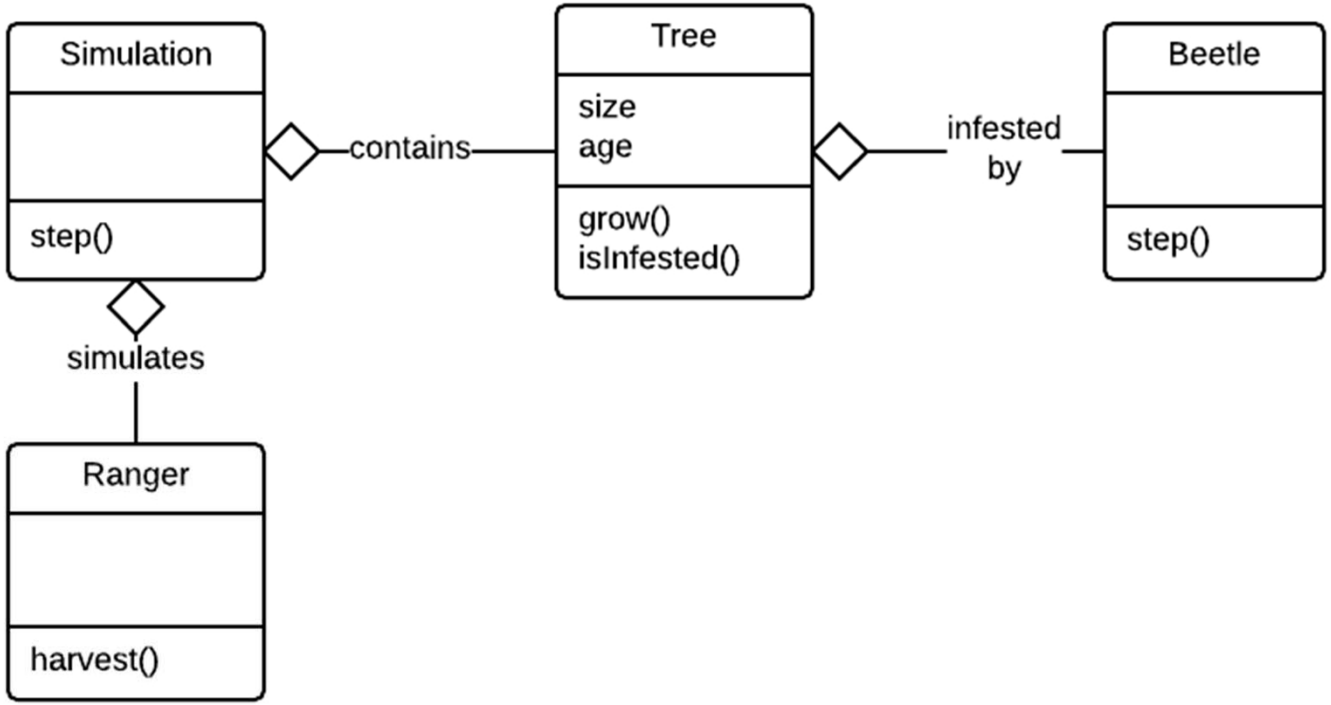
- 問題：誰がAを作るのか?
- 解決策以下の**場合、クラス A のインスタンスを作成するクラス責任を B に割り当てます。**
  - BはAのオブジェクトを集約する
  - BはAのオブジェクトを含む
  - BはAのオブジェクトのインスタンスを記録する
  - BはAのオブジェクトをよく使う
  - BはAのオブジェクトを作成するための初期化データを持っている。

- 多ければ多いほどよい。
  - BはAのオブジェクトを集約、または含む
- 重要なアイデアだ：作成者はとにかく参照を保持する必要があり、作成されたオブジェクトを頻繁に使用する。

クリエイター（GRASP）

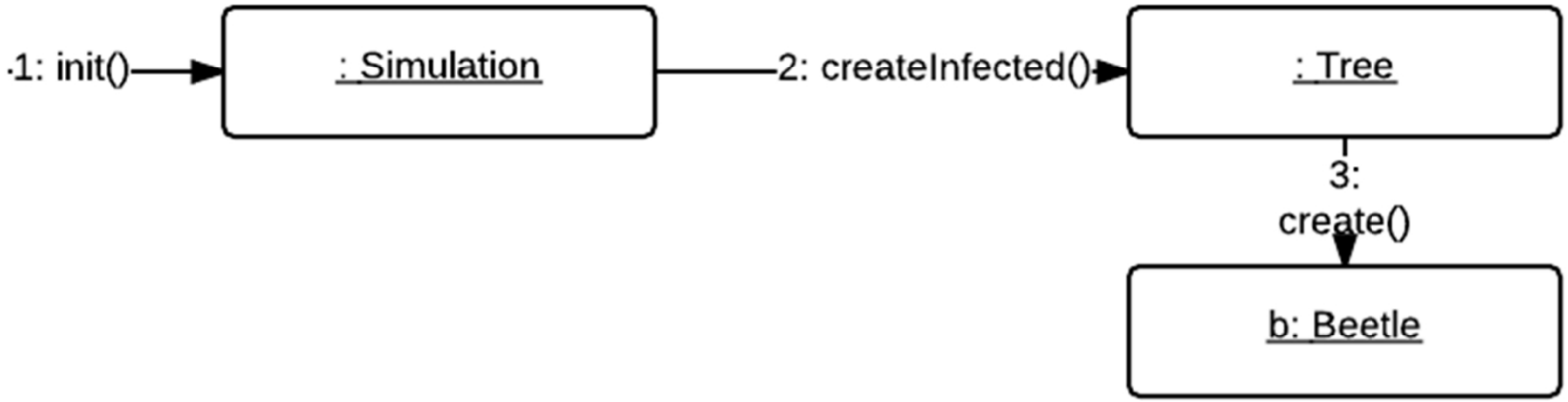
- Beetleのオブジェクトを**作成する**

  責任は誰にあるのか? 木のオブジェクト?


Simulation
step()
contains
Tree
size
age
grow()
isInfested()
infested
by
Beetle
step()
simulates
Ranger
harvest()

クリエイター：例

- ビートル・オブジェクトの作成責任者は誰ですか?
  - クリエイター・パターンが示唆するツリー
- 相互作用図:


1: init()
: Simulation
2: createInfected()
: Tree
3:
create()
b: Beetle

クリエイター（GRASP）

- 問題：オブジェクトの作成に責任を割り当てる
  - 誰がグラフのノードを作るのか?
  - 誰がSalesItemのインスタンスを作成するのか?
  - シミュレーションの中で誰が子どもたちを作るのか?

– モノポリーゲームで誰がタイルを作るのか?

- AI? プレイヤー? メインクラス? ボード? ミープル（犬）?

クリエイターデザインの議論
目標／原則

- **低結合、高結合を**促進する
  - 参照する必要があるオブジェクトの作成を担当するクラス
  - オブジェクトを自分で作成することで、オブジェクトの作成を他のクラスに依存することを避ける。
- **進化可能性を**促進する（変化のためのデザイン）
  - オブジェクトの作成は非表示、ローカルで置き換え可能

- 反対：特別な方法でオブジェクトを作成しなければならないこともある。
  - 複雑な初期化
  - 異なる状況で異なるクラスをインスタンス化する
  - *それなら、**結束は**創造を別のオブジェクトに置くことを提案する*

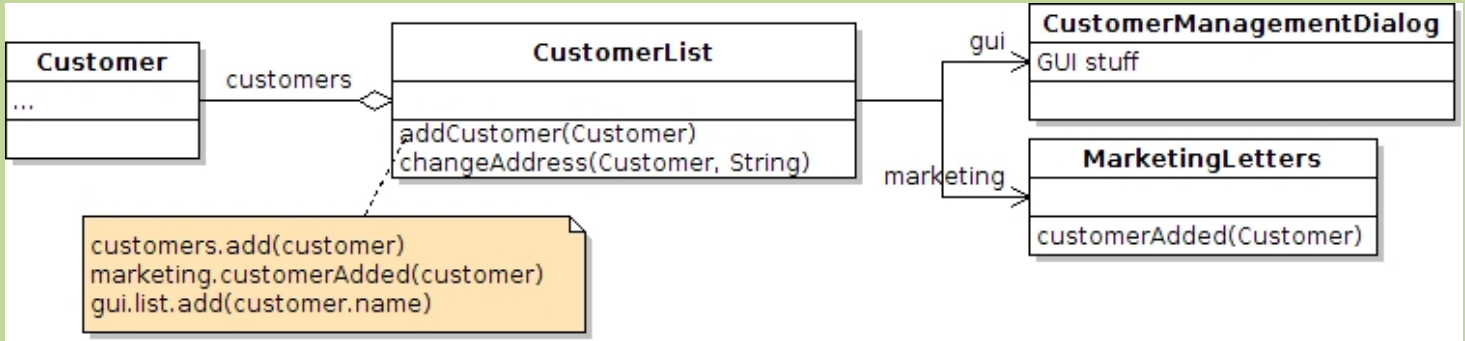
- ビルダー、ファクトリーメソッドなどの *デザインパターン*を見る

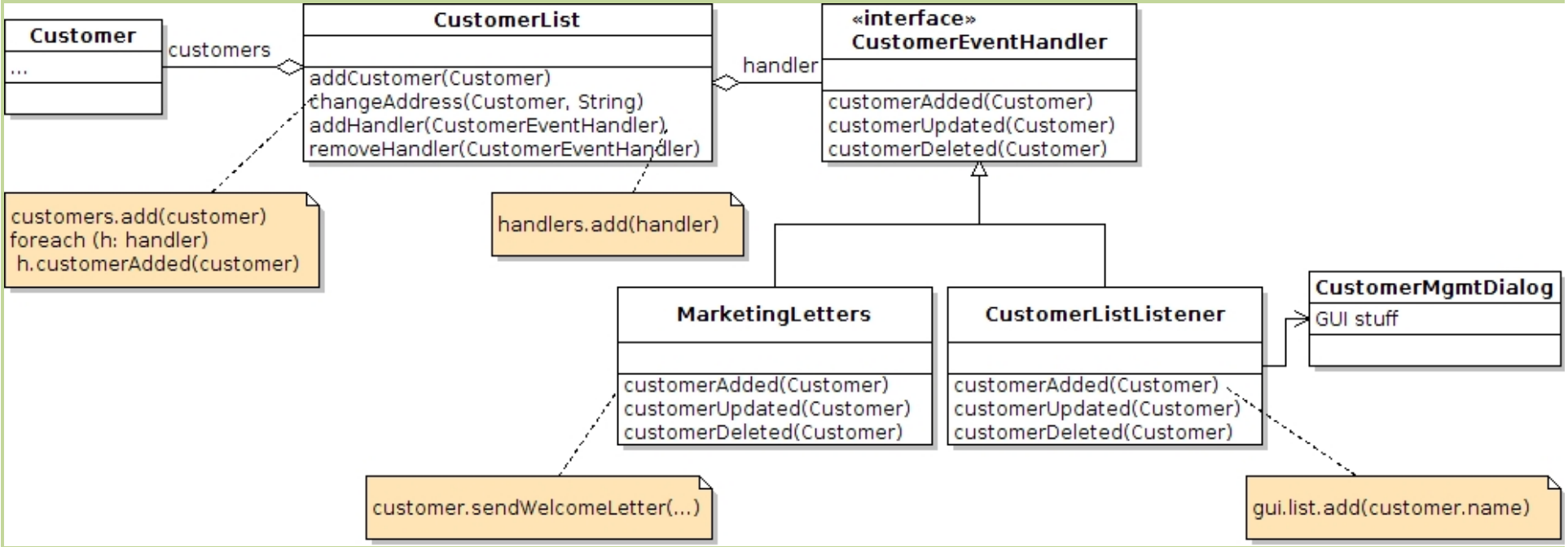
持ち帰りメッセージ

- デザインは品質属性によって左右される
  - 進化性、分離開発、再利用、パフォーマンス、…
- 設計原則は、品質を達成するためのガイダンスを提供する
  - 低結合性、高結合性、高対応性、……。

- GRASPの設計ヒューリスティックは、これらの原則を促進する。
  - クリエイター、エキスパート、コントローラー、…

どのデザインが優れているか? デザインの目標、原則で議論する、あなたが知っているヒューリスティクスやパターン





* 旧中間問題